#5

CERTIFICATE OF MAILING

I hereby certify that this paper or, if this paper is a transmittal letter, every other paper or fee referred to therein, is being deposited with the U.S. Postal Service as first class mail in an envelope addressed to Commissioner of Patents & Trademarks, Washington, DC 20231, on

October 24, 2000 (Date of Deposit)

10/24/00 Date

L Beck Name

PLEASE CHARGE ANY DEFICIENCY UP TO $300.00 OR CREDIT ANY EXCESS IN THE FEES DUE WITH THIS DOCUMENT TO OUR DEPOSIT ACCOUNT NO. 04-0100

Docket No.: **6727/0H560**

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In re Application of: **Daniel GEIST, Orna GRUMBERG and Sagi KATZ**

Serial No.: 09/605,334 Art Unit: TO BE ASSIGNED

Filed: JUNE 27, 2000 Examiner: TO BE ASSIGNED

For: **IDENTIFICATION OF MISSING PROPERTIES IN MODEL CHECKING**

- -

CLAIM FOR PRIORITY

Hon. Commissioner of
Patents and Trademarks
Washington, DC 20231

Sir:

Applicant hereby claims priority under 35 U.S.C. Section 119 based on Israeli Patent Application No. 132058 filed September 23, 1999.

A certified copy of the priority document is submitted herewith.

Respectfully submitted,

Walt Thomas Zielinski
Reg. No. 18,902
Attorney for Applicant(s)

October 24, 2000

DARBY & DARBY P.C.
805 Third Avenue
New York, New York 10022
212-527-7700

M:\6920\05331\JB7373

מדינת ישראל
STATE OF ISRAEL

Ministry of Justice
Patent Office

משרד המשפטים
לשכת הפטנטים

This is to certify that annexed hereto is a true copy of the documents as originally deposited with the patent application of which particulars are specified on the first page of the annex.

זאת לתעודה כי רצופים בזה העתקים נכונים של המסמכים שהופקדו לכתחילה עם הבקשה לפטנט לפי הפרטים הרשומים בעמוד הראשון של הנספח.

CERTIFIED COPY OF PRIORITY DOCUMENT

This 29-06-2000 היום

רשם הפטנטים
Commissioner of Patents

נתאשר
Certified

חוק הפטנטים, התשכ"ז -- 1967
PATENTS LAW, 5727-1967

בקשה לפטנט
Application for Patent

C:35277

לשימוש הלשכה
For Office Use

מספר: Number	152054
תאריך: Date	23 -09- 1999
הוקדם/נדחה Ante/Post-dates	

אני, (שם המבקש, מענו -- ולגבי גוף מאוגד -- מקום התאגדותו)
I (Name and address of applicant, and, in case of body corporate-place of incorporation)

IBM CORPORATION
International Business Machines Corporation
New Orchard Road
Armonk, N.Y. 10504
U.S.A.

Galileo Technology Ltd.
Moshav Manof
D.N. Misgav 20184

גלילאו טכנולוגיה בע"מ
משוב מנוף
ד.נ. משגב 20184

(An Israeli Company)

(חברה ישראלית)

בעל אמצאה מכח By Law שמה הוא
Owner, by virtue of

of an invention, the title of which is:

(בעברית) זיהוי של תכונות חסרות בבדיקת מודלים
(Hebrew)

(באנגלית) IDENTIFICATION OF MISSING PROPERTIES IN MODEL CHECKING
(English)

hereby apply for a patent to be granted to me in respect thereof מבקש בזאת כי ינתן לי עליה פטנט

*בקשה חלוקה - Application for Division	*בקשת פטנט מוסף - Application for Patent of Addition	*דרישה דין קדימה Priority Claim		
מבקשת פטנט from Application	*לבקשה/לפטנט to Patent/Appl.	מספר/סימן Number/Mark	תאריך Date	מדינת האיגוד Convention Country
No._____ מס.	No._____ מס.			
dated_____ מיום	dated_____ מיום			
*יפוי כח: כללי/מיוחד - רצוף בזה / עוד יוגש P.O.A.: general / individual - attached / to be filed later - filed in case_____ הוגש בענין				
המען למסירת הודעות ומסמכים בישראל Address for Service in Israel: Sanford T. Colb & Co. P.O.B. 2273 Rehovot 76122				
חתימת המבקש Signature of Applicant: For the Applicant, Sanford T. Colb & Co. C:35277		היום This 23 בחודש of September שנת of the year 1999		

לשימוש הלשכה
For Office Use

טופס זה, כשהוא מוטבע בחותם לשכת הפטנטים ומושלם בספר ובתאריך ההגשה, הינו אישור להגשת הבקשה שפרטיה רשומים לעיל.
This form, impressed with the Seal of the Patent Office and indicating the number and date of filing, certifies the filing of the application, the particulars of which are set out above

זיהוי של תכונות חסרות בבדיקת מודלים

IDENTIFICATION OF MISSING PROPERTIES IN MODEL CHECKING

IBM CORPORATION
GALILEO TECHNOLOGY LTD.

גלילאו טכנולוגיה בע"מ

C: 35277

IDENTIFICATION OF MISSING PROPERTIES IN MODEL CHECKING

FIELD OF THE INVENTION

The present invention relates generally to design automation and verification, and specifically to hardware verification of integrated circuit design.

BACKGROUND OF THE INVENTION

Hardware verification is currently the bottleneck and the most expensive task in the design of a semiconductor integrated circuit. Model checking is a method of formal verification that is gaining in popularity for this purpose. The method is described generally by Clarke et al. in *Model Checking* (MIT Press, 1999), which is incorporated herein by reference.

To perform model checking of the design of a device, a verification engineer reads the definition and functional specifications of the device and then, based on this information, writes a set of properties $\{\phi\}$ (also known as a specification) that the design is expected to fulfill. The properties are written in a suitable specification language for expressing temporal logic relationships between the inputs and outputs of the device. Such languages are commonly based on Computation Tree Logic (CTL). A hardware model M (also known as an implementation) of the design, which is typically written in a hardware description language, such as VHDL or Verilog, is then tested to ascertain that the model satisfies all of the properties in the set, i.e., that $M \models \phi$, under all possible input

sequences. Such testing is a form of reachability analysis.

Model checking is preferably carried out automatically by a symbolic model checking program, such as SMV, as described, for example, by McMillan in *Symbolic Model Checking* (Kluwer Academic Publishers, 1993), which is incorporated herein by reference. A number of practical model checking tools are available, among them RuleBase, developed by IBM, which is described by Beer et al. in "RuleBase: an Industry-Oriented Formal Verification Tool," in *Proceedings of the Design Automation Conference DAC'96* (Las Vegas, Nevada, 1996), which is incorporated herein by reference.

As hardware devices grow larger and more complex, the set of properties needed for model checking becomes unwieldy. The verification engineer has no systematic way to be sure of whether the property set is complete, in the sense of covering all possible states and transitions that may occur in the model. If the property set is incomplete, a bug in the design may go undetected. The engineer may therefore continue to add more and more properties indefinitely, never knowing whether the set is yet sufficient or not.

Coverage metrics have been applied in various fields of simulation-based verification in order to measure and improve the completeness with which a given simulation tool represents the actual behavior of a target system. An application of such a metric to model checking is described by Hoskote et al. in "Coverage Estimation for Symbolic Model Checking," in *Proceedings of the Design Automation Conference DAC'99* (IEEE Computer Society Press, 1999), which is

incorporated herein by reference. The authors present a method for estimating whether a set of properties is sufficient to cover all possible states of a model. They note, however, that their method cannot point out functionality that may be missing in the model, nor can it ensure that all possible paths between the states are covered. They indicate that "path coverage would be an ideal coverage metric because it can provide coverage of actual executions of the circuit over time." The authors consider that by comparison with state coverage, "path coverage is a much more intractable problem."

SUMMARY OF THE INVENTION

It is an object of some aspects of the present invention to provide improved methods and systems for design verification.

It is a further object of some aspects of the present invention to provide improved methods and metrics for analyzing the coverage of a set of properties used in model checking, and in particular to provide methods for analyzing path coverage.

In preferred embodiments of the present invention, a specification, consisting of properties, is generated to verify a given implementation model of a target system, and a tableau is constructed corresponding to the properties. Such a tableau is defined as a finite state machine that satisfies all of the properties in the specification. The states and transitions of the tableau are compared to those of the model to ascertain that there is full correspondence between the possible states and transitions of the model and those of the tableau. To the extent that there are no substantive differences, it is concluded that the set of properties fully specifies the model.

In some preferred embodiments of the present invention, the tableau is compared to the model by inputting the tableau to a model checking program, such as SMV, along with the given model. If for every possible combination of inputs, the tableau gives exactly the same set of outputs as the model, then the specification of the properties is complete. On the other hand, if a difference occurs for some input, it means that the specified properties are insufficient and/or that there is an error in the model. The fact that the outputs of the tableau exactly correspond to

those of the model indicates that for every reachable state of the tableau, there is a corresponding state in the model, and for every possible transition in the tableau, there is a corresponding transition between the corresponding states in the model. It also means that there are no excess transitions or spurious initial conditions in the tableau that would allow transitions to be made among states in a way that would not be possible in the model.

The methods of the present invention thus inform the user when the specification of properties is complete or, alternatively, provide an indication as to where there may be flaws in the correspondence between the specification and the model. Such flaws typically point either to a state or transition in the tableau that is not implemented in the model, thus warning either that the specification does not adequately constrain the model, or that the model has failed to implement a meaningful state or transition of the target system. By comparison, the method of Hoskote et al. is limited to finding an estimation of state coverage, and not transitions or exact state coverage, and therefore cannot provide a conclusive indication that the set of properties is complete.

There is therefore provided, in accordance with a preferred embodiment of the present invention, a method for verification, including:

providing an implementation model, which defines model states of a target system and model transitions between the model states;

providing a specification of the target system, including properties that the system is expected to obey;

creating a tableau from the specification, the tableau defining tableau states with tableau transitions between the tableau states in accordance with the properties; and

comparing the tableau transitions to the model transitions to determine whether a discrepancy exists therebetween.

In a preferred embodiment, creating the tableau includes defining a finite state machine using a hardware description language, wherein the implementation model has model inputs and outputs, and wherein defining the finite state machine includes describing a virtual device having inputs and outputs corresponding to the model inputs and outputs of the implementation model. Preferably, comparing the transitions includes performing a reachability analysis using both the implementation model and the tableau while providing identical inputs to the inputs of both the implementation model and the tableau, and verifying that the outputs are always identical. Most preferably, performing the reachability analysis includes comparing the model and the tableau automatically using a model checker and providing evidence of a tableau transition that is not implemented in the model.

In another preferred embodiment, comparing the tableau transitions includes associating model transitions with corresponding tableau transitions, wherein associating the transitions includes defining a reachable simulation preorder relating the model and the tableau. Preferably, associating the transitions includes finding a tableau transition that is not implemented in the model and, most preferably, deriving

an indication, based on the unimplemented transition, that the specification is not complete with respect to the model. Alternatively or additionally, finding the tableau transition that is not implemented in the model includes deriving an indication, based on the unimplemented transition, that a transition of the target system is missing in the model.

Preferably, the method includes associating model states with corresponding tableau states. Further preferably, associating the model states with the corresponding tableau states includes finding a tableau state that is not implemented in the model and deriving an indication, based on the unimplemented state, that the specification is not complete with respect to the model. Alternatively or additionally, finding the tableau state that is not implemented in the model includes deriving an indication, based on the unimplemented state, that a state of the target system is missing in the model. Further alternatively or additionally, associating the model states with the corresponding tableau states includes finding multiple model states corresponding to a single tableau state.

Preferably, creating the tableau includes creating a reduced tableau from which one or more redundant states have been eliminated.

Further preferably, comparing the transitions includes verifying that the specification is a complete and correct description of the implementation model responsive to the comparison.

There is also provided, in accordance with a preferred embodiment of the present invention, a verification processor, which is configured to receive an implementation model, defining model states of a

target system and model transitions between the model states, and to receive a specification of the target system, including properties that the system is expected to obey, and which is operative to create a tableau from the specification, the tableau defining tableau states with tableau transitions between the tableau states in accordance with the properties, and to compare the tableau transitions to the model transitions to determine whether a discrepancy exists therebetween. Preferably, the processor is operative to perform model checking of the implementation model.

There is further provided, in accordance with a preferred embodiment of the present invention, a computer software product for verification of a specification of a target system, which specification includes properties that the system is expected to obey, by comparison with an implementation model, which defines model states of the target system and model transitions between the model states, the product including a computer-readable medium having computer program instructions recorded therein, which instructions, when read by a computer, cause the computer to create a tableau from the specification, the tableau defining tableau states with tableau transitions between the tableau states in accordance with the properties, and to compare the tableau transitions to the model transitions to determine whether a discrepancy exists therebetween.

Preferably, the program instructions cause the computer to compare the tableau with the model by running a reachability analysis using both the implementation model and the tableau while providing identical inputs to the inputs of both the

implementation model and the tableau, and verifying that the outputs are always identical. Most preferably, the reachability analysis is performed using an automatic model checker, and the instructions cause the computer to verify that the specification is a complete description of the implementation model.

There is additionally provided, in accordance with a preferred embodiment of the present invention, a method for verification, including:

providing an implementation model, which defines model states of a target system and model transitions between the model states;

providing a specification of the target system, including properties that the system is expected to obey;

creating a tableau from the specification, the tableau defining tableau states with tableau transitions between the tableau states in accordance with the properties; and

comparing the model and the tableau by inputting the model and the tableau to an automatic model checking program.

Preferably, comparing the model and the tableau includes providing evidence of a transition or state in the tableau that is not implemented in the model, most preferably in the form of a counter-example indicative of the unimplemented transition or state.

There is moreover provided, in accordance with a preferred embodiment of the present invention, model checking apparatus, which is configured to receive an implementation model, defining model states of a target system and model transitions between the model states, and to receive a specification of the target system,

including properties that the system is expected to obey, and which is operative to create a tableau from the specification, the tableau defining tableau states with tableau transitions between the tableau states in accordance with the properties, and to compare the tableau to the model by inputting the model and the tableau to an automatic model checking program.

There is furthermore provided, in accordance with a preferred embodiment of the present invention, a computer software product for verification of a specification of a target system, which specification includes properties that the system is expected to obey, by comparison with an implementation model, which defines model states of the target system and model transitions between the model states, the product including a computer-readable medium having computer program instructions recorded therein, which instructions, when read by a computer, cause the computer to create a tableau from the specification, the tableau defining tableau states with tableau transitions between the tableau states in accordance with the properties, and to compare the tableau to the model by inputting the model and the tableau to an automatic model checking program.

The present invention will be more fully understood from the following detailed description of the preferred embodiments thereof, taken together with the drawings in which:

BRIEF DESCRIPTION OF THE DRAWINGS

Fig. 1 is a schematic pictorial illustration showing a system for model checking, in accordance with a preferred embodiment of the present invention;

Fig. 2 is a block diagram that schematically illustrates an implementation model used in model checking and a corresponding tableau of properties, in accordance with a preferred embodiment of the present invention;

Fig. 3 is a state diagram that schematically illustrates a finite state machine corresponding to the tableau of Fig. 2, in accordance with a preferred embodiment of the present invention;

Fig. 4 is a flow chart that schematically illustrates a method for verifying the correctness of a set of properties generated for the purpose of model checking, in accordance with a preferred embodiment of the present invention; and

Fig. 5 is a flow chart that schematically illustrates another method for verifying the correctness of a set of properties generated for the purpose of model checking, in accordance with a preferred embodiment of the present invention.

DETAILED DESCRIPTION OF PREFERRED EMBODIMENTS

Fig. 1 is a schematic pictorial illustration of a system 20 for model checking, in accordance with a preferred embodiment of the present invention. System 20 typically comprises a verification processor 22, typically a general-purpose computer workstation running suitable model checking software, such as the above-mentioned IBM RuleBase, under the control of a verification engineer 24. The system receives a hardware implementation model 26 of a target system or device 30 in development. Engineer 24 prepares a specification of properties 28, for use in model checking of model 26. The completeness and correctness of the specification are verified by system 20 using methods described in detail hereinbelow.

Reference is now made to Fig. 2, which is a block diagram representing a model of a target hardware device 40, in this case a simple two-port synchronous arbiter, used hereinbelow to exemplify a method for verifying property set 28, in accordance with a preferred embodiment of the present invention. Device 40 has two request inputs 42, labeled REQ0 and REQ1, and two acknowledge outputs 44, ACK0 and ACK1. The assertion of ACKi is a response to the assertion of REQi. Initially, both outputs of the arbiter are inactive. At any time, at most one acknowledge output may be active. The arbiter grants one of the active requests in the next cycle, and uses a round robin algorithm in case both request inputs are active. In the case of simultaneous assertion (i.e. both requests are asserted and were not asserted in the previous cycle), REQ0 has priority in the first simultaneous assertion occurrence. In any subsequent occurrence of

simultaneous assertion the priority rotates with respect to the previous occurrence.

An implementation of device 40 in the SMV language is presented below in Table I:

TABLE I

```
1) var
2)   req0; req1; ack0; ack1; robin : boolean;
3) assign
4)   init(ack0) := 0;
5)   init(ack1) := 0;
6)   init(robin):=0;
7) next(ack0)   := case
8)   !req0          : 0;       - No request results no
                                    ack
9)   !req1          : 1;       - A single request
10)  !ack0&!ack1    :!robin;   - Simultaneous requests
                                    assertions
11)  1              :!ack0;    - Both requesting, toggle
                                    ack
12) esac;
13) next(ack1) := case
14)  !req1          : 0;       - No request results no
                                    ack
15)  !req0          : 1;       - A single request
16)  !ack0&!ack1    : robin;   - Simultaneous assertion
17)  1              : !ack1;   - Both requesting, toggle
                                    ack
18) esac;
19) next(robin) :=if req0&req1&!ack0&!ack1 then !robin
20)                 else robin endif; - Two
                                    simultaneous request
                                    assertions
```

From the functional specification of arbiter 40 given above, the following temporal formulas are derived that describe the properties of the device:

1. The initial state is $\neg ack0 \wedge \neg ack1$.
2. At all times, mutual exclusion holds, i.e., $\neg ack0 \vee \neg ack1$ (property $\phi 1$).
3. At all times one of the following properties should hold:
   a) No requests followed by no acknowledge (property $\phi 2$).
   b) A single request (when the other request is not active) is served in the following cycle (properties $\phi 3$ and $\phi 4$).
   c) A request active while the alternate channel is being served will be served in the following cycle (properties $\phi 5$ and $\phi 6$).
   d) When a cycle with no active request is followed by a cycle with two active requests, the result will be as follows -
      - The first such occurrence will result in acknowledgment to channel 0 ($\phi 0$).
      - Each subsequent occurrence will result in acknowledgment of the channel that was not acknowledged in the previous occurrence ($\phi 7$ and $\phi 8$).

   The behavior of the arbiter under these conditions (no active request followed by two active requests) is governed by the non-observable variable "robin," as defined in Table I.

The above formulas correspond to the properties $\phi 0$, $\phi 1$, …, $\phi 8$ listed symbolically below in Table II, which are a complete specification of arbiter 40 written in the form of a safety formula ψ in Universal Computation Tree Logic (known as ACTL). ACTL is a branching-time temporal logic. It is described in detail by Grumberg et al. in "Model Checking and Modular Verification," in *ACM Transactions on Programming Languages and Systems* **16**(3) (1994), pp. 843-871, which is incorporated herein by reference. As the variable "robin" is not observable, it does not appear explicitly in the properties in Table II.

TABLE II

$\psi = \neg ack0 \wedge \neg ack1 \wedge$

$\mathbf{A}[(\neg req0 \vee \neg req1 \vee ack0 \vee ack1)\mathbf{W}$

$(req0 \wedge req1 \wedge \neg ack0 \wedge \neg ack1 \wedge \mathbf{AX} ack0)] \wedge$ - ϕ_0

$\mathbf{AG}($

$(\neg ack0 \vee \neg ack1) \wedge$ - ϕ_1

$(\neg req0 \wedge \neg req1 \rightarrow \mathbf{AX}(\neg ack0 \wedge \neg ack1)) \wedge$ - ϕ_2

$(req0 \wedge \neg req1 \rightarrow \mathbf{AX}\ ack0) \wedge$ - ϕ_3

$(\neg req0 \wedge req1 \rightarrow \mathbf{AX}\ ack1) \wedge$ - ϕ_4

$(req1 \wedge ack0 \rightarrow \mathbf{AX}\ ack1) \wedge$ - ϕ_5

$(req0 \wedge ack1 \rightarrow \mathbf{AX}\ ack0) \wedge$ - ϕ_6

$(req0 \wedge req1 \wedge \neg ack0 \wedge \neg ack1 \rightarrow \mathbf{AX}(ack0 \rightarrow$

$\mathbf{A}[(\neg req0 \vee \neg req1 \vee ack0 \vee ack1)\mathbf{W}$

$(req0 \wedge req1 \wedge \neg ack0 \wedge \neg ack1 \wedge \mathbf{AX} ack1)])) \wedge$ - ϕ_7

$(req0 \wedge req1 \wedge \neg ack0 \wedge \neg ack1 \rightarrow \mathbf{AX}(ack1 \rightarrow$

$\mathbf{A}[(\neg req0 \vee \neg req1 \vee ack0 \vee ack1)\mathbf{W}$

$(req0 \wedge req1 \wedge \neg ack0 \wedge \neg ack1 \wedge \mathbf{AX} ack0)]))$ - ϕ_8

This representation uses the temporal operators X ("next state"), W ("weak until," i.e., remains true until), and G ("globally"), along with the quantifier A ("for all paths"). It is noted that $AG\phi \equiv A[\phi W false]$. Further details regarding ACTL safety formulas in the context of the present invention are presented in Appendix A.

Fig. 3 is a state diagram that schematically illustrates a tableau 50, or state machine, corresponding to the properties of the safety formula in Table II, in accordance with a preferred embodiment of the present invention. The tableau is preferably a "reduced tableau," as defined in Appendix A, which is most preferably constructed automatically, using a computer program that receives the properties as its input and implements the tableau construction algorithm listed in the appendix. The tableau is used, as described further hereinbelow, to verify that the properties completely cover the states and transitions of the device model defined in Table I.

Tableau 50 comprises six state groups 52, 54, 56, 58, 60 and 62. Each of the groups includes a number of states 64 that correspond to a particular output condition of device 40, i.e., in groups 52 and 60, ack0 is asserted; in groups 54 and 62, ack1 is asserted; and in groups 56 and 58, neither output is asserted (!ack0 !ack1). In state groups 52, 54 and 56, the non-observable variable robin = 0, whereas in groups 58, 60 and 62, robin = 1. As indicated by an arrow 66, operation of device 40 begins in group 56, with !ack0, !ack1 and robin = 0. Transitions from one state to

another depend on the choice of inputs, which are marked in each state 64. Thus, for example, when req1 is asserted in state group 52, a transition is invoked to group 54, in which ack1 is asserted.

Tableau 50 is a sort of virtual device, corresponding to actual target device 40. Appendix B accordingly contains source code in VHDL representing a tableau similar to tableau 50 as a device model. In preferred embodiments of the present invention, this virtual device is tested to determine whether its states and transitions correspond exactly to those of the actual device, or equivalently whether the behavior of the virtual device under all possible combinations of input conditions is identical to that of the model of the actual device. If differences are found, they are then indicative either that the tableau (and hence the specified properties) are incomplete or that the model itself is incomplete. Methods and criteria for testing tableau 50 are described further hereinbelow.

Fig. 4 is a flow chart that schematically illustrates a method for verifying a specification of model checking properties by comparing tableau 50 with device 40, in accordance with a preferred embodiment of the present invention. The method begins with preparation of the specification, such as the formula ψ listed in Table II, and checking the device model M to ensure that the model satisfies the specification, i.e., that $M \models \psi$. The specification is then used to construct the tableau. As shown in Fig. 2, tableau 50 as a virtual device model has virtual inputs 72 and outputs 74, corresponding respectively to inputs 42 and outputs 44 of implementation model 40 of the target

device. For the purpose of testing the tableau against the implementation model, the tableau inputs are labeled REQ0_SPEC and REQ1_SPEC, and the tableau outputs are labeled similarly, ACK0_SPEC and ACK1_SPEC.

These input and output labels are inserted in a representation of the tableau, preferably in the form of suitable program code, as listed in Appendix B. IBM RuleBase, as described hereinabove, is capable of translating the VHDL code in Appendix B into the SMV language used by model checkers. In the example shown in Appendix B, the model of device 40 is simplified, relative to the definition in Tables I and II, in that the non-observable variable "robin" is not used. Instead, ack0 always receives priority when a state in which there is no active request is followed by two active requests.

In order to test the tableau against the device model, a new model is created, combining the original implementation model and the virtual device model of the tableau. Inputs 72 of tableau 50 are tied to the corresponding inputs 42 of implementation model 40, so that the implementation model and the tableau model will receive the same input signals. The new, combined model is then input to an automatic model checking program, such as SMV. The model checker is asked to verify that the following properties regarding the combined model outputs are always true of the combined model:

```
ACK0==ACK0_SPEC
ACK1==ACK1_SPEC
```

Alternatively, in certain cases, the two outputs may be checked separately, rather than in a single pass of the model checker, using separate tableaux corresponding to subsets of the specification properties that influence the particular outputs.

If the above-mentioned properties of the combined model outputs are confirmed, tableau 50 is assured of representing a complete and correct specification of device 40. If the tableau specification is not sufficiently detailed, then the tableau outputs ACKi_SPEC will not be as constrained as the model outputs ACKi, and there will be some combination of inputs under which ACKi_SPEC will have two possible values when ACKi can have only one. This outcome will generally lead engineer 24 to conclude that an additional constraint is required, i.e., that a further property is needed in the specification. Typically, a suitable property is added and the specification is re-checked, repeating the steps described above. If now ACKi==ACKi_SPEC, then the specification can be considered complete and correct. Alternatively, it may turn out that evaluation of the model and specification in this manner will lead engineer 24 to conclude that there is an error in the implementation model, such as a missing state or transition, which causes the outputs of the model and the tableau differ. It is a further advantage of automatic model checking programs that they provide evidence of such errors, in the form of "counter-examples," that assist the engineer in identifying and correcting the error.

Fig. 5 is a flow chart that schematically illustrates another method for verifying a model checking specification, in accordance with a preferred

embodiment of the present invention. The method begins, like the method of Fig. 4, with preparation of a specification of model properties and model checking to determine that the model satisfies the specification. The specification is then used to create a corresponding tableau, which is evaluated against the device implementation model. The method of Fig. 5 differs from the method of Fig. 4 in the manner in which the tableau is evaluated, as described hereinbelow. Whereas the method of Fig. 4 is useful primarily in checking deterministic models, the method of Fig. 5 can be used for substantially any model, including non-deterministic models.

In order to evaluate the tableau against the model, a simulation preorder, SIM, is calculated for the model and the tableau. SIM is a relation between the model M and the tableau T ($SIM \subseteq M \times T$) containing pairs of states (s,s') in M and T, respectively. SIM satisfies the following requirements:

- For every initial state s_0 of M, there is an initial state s_0' of T, such that (s_0,s_0') belongs to SIM.
- For every pair of states (s,s') in SIM, state s is characterized by the same set of atomic propositions as s' (i.e., the same values of the variables ACKi and REQi in the example of device 40).
- For every state-to-state transition from state s, there is a corresponding transition for state s' (although not necessarily a one-to-one correspondence).

A formal definition and method for computation of SIM are presented in Appendix C hereinbelow.

Based on the simulation preorder SIM, a reachable simulation preorder for M and T, ReachSIM is determined. T may contain paths by which a state s is reached from an initial state, which do not have corresponding permissible paths in M. ReachSIM $\subseteq$ SIM contains only pairs of states (s,s') characterized in that states s and s' are reached by corresponding paths π and π' from corresponding initial states. A path $\pi = s_0, s_1, \ldots, s$, and a path $\pi' = s_0', s_1', \ldots, s'$ are considered to correspond if every pair of states (s_i, s_i') along the paths belongs to SIM. Details of the computation of ReachSIM are similarly presented in Appendix C.

ReachSIM thus identifies a set of states in T having transitions that correspond to the actual transitions in M. There may yet be, however, states or transitions in T that do not have corresponding states or transitions in M, meaning that the specification does not constrain the implementation model tightly enough, so that one or more additional properties are needed. Such states and transitions are referred to herein as being "unimplemented." There may likewise be states in T that correspond simultaneously to two or more states in M. Such discrepancies are detected using ReachSIM to evaluate the following criteria, either serially or in parallel:

- Unimplemented start states: These are initial tableau states that have no corresponding initial states in the model (and therefore are absent from ReachSIM). The existence of an

unimplemented start state indicates either that the specification does not adequately constrain the start states, or that the model is lacking a required initial state.

- Unimplemented states: The existence of a state anywhere in the tableau that is not included in ReachSIM indicates either that the specification is lacking in constraints or that a meaningful state of the device is not implemented in the model.
- Unimplemented transitions: These are transitions between states of the tableau for which there is no corresponding transition in the model. The states belong to ReachSIM, so that they have corresponding states in the model, which are reached by corresponding paths. The existence of an unimplemented transition indicates either that the specification is not tight enough or that a required transition, between reachable implementation states, was not implemented in the model.
- Many-to-one mapping: In this case, there may be a tableau state to which multiple implementation states are mapped, i.e., a state s which is paired in ReachSIM with at least two different model states s_i' and s_j'. The existence of a many-to-one state indicates either that the specification is not sufficiently detailed, or that the implementation contains redundancies.

If the first three of these four criteria return empty results, then T is a complete specification of M. Any dissimilarity between the tableau and the

implementation will result in a non-empty result. Preferably, the tableau T that is used in this method is a reduced tableau, as defined in Appendix A, since traditional (non-reduced) tableaux typically contain redundancies, which are removed in the reduced tableau. If the first three of the criteria above hold (i.e., return empty results), T and M are bisimilar, and the fourth criterion is not necessary to establish the completeness of T. It may, however, indicate that there are redundancies in the implementation.

As noted hereinabove, verification of specification properties vis-a-vis the implementation model, using any of the methods described herein, is preferably carried out using software for this purpose running on processor 22. Software for use in such verification is preferably supplied as component of a simulation and model checking software package. Alternatively, the software for tableau construction and verification is provided as an independent software package. In either case, the software may be conveyed to processor 22 in intangible form, over a network, for example, or on tangible media, such as CD-ROM.

Although preferred embodiments are described hereinabove with reference to certain methods and languages used in model checking, it will be understood that the application of the present invention is not limited to any particular language or method of implementation. Those skilled in the art will appreciate that the principles of the present invention may similarly be used in other areas of verification, not only for electronic devices, but also in verification of other types of target systems, as well, for example, transportation systems or complex valve

manifolds. It will thus be understood that the preferred embodiments described above are cited by way of example, and the full scope of the invention is limited only by the claims.

APPENDIX A

REDUCED TABLEAU FOR ACTL - DEFINITION AND ALGORITHM

1 Reduced Tableau for ACTL

In this section we define a *reduced tableau* for the subset of ACTL safety formulas. We follow the definition of the reduced tableau for LTL presented in [2]. A tableau is a special form of a Kripke structure, consisting of states labeled with atomic propositions and transitions between the states. As is often the case with tableaux for temporal logics (e.g. [1]), a state of the tableau consists of a set of formulas that are supposed to hold along all paths leaving the state. Unlike typical tableaux, however, the formulas in the states of the reduced tableau are interpreted over a three-valued domain. Thus, a state may include a formula or its negation, or none of the two. If the latter occurs, it reflects a "don't care" situation, i.e., the formula may be either true or false in the state.

Similarly to [1], we wish the reduced tableau for a formula ψ to satisfy ϕ. Furthermore, it should be greater by the *simulation preorder* [3] than any Kripke structure that satisfies ψ. In order to achieve these goals we will adapt both definitions of $\models$ and simulation preorder to be applicable to three-valued structures. Below we present the formal definitions of the tableau and of the adapted relations.

Let AP_ψ be the set of atomic propositions in an ACTL formula ψ .

Definition 1.1 (sub-formulas) *The set of* sub-formulas *of ψ is defined recursively as follows :*

1. $sub(p) = \{p\}$ *and* $sub(\neg p) = \{\neg p\}$, *if* $p \in AP_\psi$
2. $sub(\varphi) = \{\varphi\} \cup sub(g_1) \cup sub(g_2)$, *if* $\varphi = g_1 \wedge g_2$ *or* $\varphi = g_1 \vee g_2$.
3. $sub(AXg_1) = \{AXg_1\} \cup sub(g_1)$
4. $sub(A[g_1 W g_2]) = \{A[g_1 W g_2],\ AXA[g_1 W g_2]\} \cup sub(g_1) \cup sub(g_2)$

We will distinguish between α-formulas and β-formulas, which are conjunctions and disjunctions, respectively. In the reduced tableau, if a state contains a conjunction, then it also contains its two conjuncts. On the other hand, if it contains a disjunction, it will usually contain only one of the disjunct, leaving the other as "don't care".

Definition 1.2 (α-formula) *A formula $g \in sub(\psi)$ is an α-formula if $g = g_1 \wedge g_2$. In this case we define a set of formulas $k(g) = \{g_1, g_2\}$.*

Definition 1.3 (β-formula) *A formula $g \in sub(\psi)$ is a β-formula if :*

1. $g = g_1 \vee g_2$, *in which case* $k_1(g) = \{g_1\}$ *and* $k_2(g) = \{g_2\}$.
2. $g = A[g_1 W g_2]$, *in which case* $k_1(g) = \{g_2\}$ *and* $k_2(g) = \{g_1, AXA[g_1 W g_2]\}$.

Definition 1.4 (A particle) *A set of formulas P is a particle if:*

1. $P \subseteq sub(\psi)$
2. $p \in P \rightarrow \neg p \notin P$
3. $\neg p \in P \rightarrow p \notin P$
4. *For every α-formula $g \in sub(\psi)$, $g \in P$ iff $k(g) \subset P$*
5. *For every β-formula $g \in sub(\psi)$, $g \in P$ iff either $k_1(g) \subset P$ or $k_2(g) \subset P$ or both.*

Definition 1.5 (Implied successor) *A formula g is an* implied successor *of a particle P if $AXg \in P$. We denote by $imps(P)$ the set of implied successors of P, i.e., $imps(P) = \{g | AXg \in P\}$*

Note that if P does not include any formula of the form AXg then $imps(P) = \{\}$. The particle $\{\}$ means that the state reached has no commitments to satisfy any of the formulas. Thus, it may be the start of any possible paths. Furthermore, it may simulate any state. We later see that the only son of particle $\{\}$ is the particle $\{\}$ itself.

Definition 1.6 (α-closed) *A set of formulas B is α-closed, if for every α-formula $r \in sub(\psi)$, $r \in B$ iff $k(r) \subset B$. The α-closure of a set B is the smallest α-closed set containing B.*

Definition 1.7 (β-closed) *A set of formulas B is β-closed, if for every β-formula $r \in sub(\psi)$, $r \in B$ iff either $k_1(r) \subset B$ or $k_2(r) \subset B$ (or both).*

Function : *RemoveRedundant*
function *RemoveRedundant*(Set of particles) returns : Set of particles
This function receives a set of particles, and returns a set such that : For each one of the elements in the set, The element will stay in the set if it does not include any other element in the set, and will be removed otherwise.
Function : $cover_p$
recursive function $cover_p(B$: Set of formulas) returns : Set of particles
if B is not locally consistent then return $\{\}$ – no particles contain B
if B is not α-closed then return $cover_p(\alpha\text{-closure}(B))$
if there exists some β-formula $r \in sub(\psi)$ such that $r \notin B$ but $k_1(r) \in B$ or $k_2(r) \subseteq B$ then return $cover_p(B \cup \{r\})$
if there exists some β-formula $r \in B$ but both $k_1(r)$ and $k_2(r)$ are not in B
then return $RemoveRedundant(cover_p(B \cup \{k_1(r)\}) \cup cover_p(B \cup \{k_2(r)\}))$
return B – B is a particle
end function

Definition 1.8 ($Successors_p(P)$) $Successors_p(P) = cover_p(imps(P))$.

We now describe an iterative algorithm PART_TAB that produces the tableau $\tau(\psi) =< S_\tau, S_{\tau 0}, R_\tau, L_\tau >$ for ψ.
Algorithm : PART_TAB
$S_{\tau 0} := cover_p(\{\psi\})$
$S_\tau := S_{\tau 0}$
$R_\tau := \emptyset$
Mark all particles in S_τ as unprocessed
For each unprocessed particle P in S_τ do
$S := successors_p(P)$;
For each $Q \in S$
Add (P, Q) to R_τ;
Define $L(P) = P \cap (\{p | p \in AP_\psi\} \cup \{\neg p | p \in AP_\psi\}$
If $Q \notin S_\tau$;
Add Q to S_τ;
Mark Q as unprocessed;
end for

Mark P as processed;
end for
end for

Note that a state is labeled by propositions from AP and by their negations. Since a state is a particle, it will never contain both a proposition and its negation, but it may contain none of them.

References

[1] O. Grumberg and D.E. Long. Model checking and modular verification. *ACM Trans. on Programming Languages and Systems*, 16(3):843–871, 1994.

[2] Z. Manna and A. Pnueli. *Temporal verifications of Reactive Systems - Safety.* Springer-Verlag, 1995.

[3] R. Milner. An algebraic definition of simulation between programs. In *In proceedings of the 2nd International Joint Conference on Artificial Intelligence*, pages 481–489, September 1971.

APPENDIX B

The following is a program listing in VHDL describing a tableau that corresponds generally to the properties of arbiter 40 listed in Table II. As noted hereinabove, however, the listing below is based on a simplified model without the non-observable variable "robin." In place of the output names ACK0_SPEC and ACK1_SPEC shown in Fig. 2, the names ack0_new and ack1_new are used in the listing; and req0 and req1 in the listing correspond respectively to REQ0_SPEC and REQ1_SPEC.

```
library ieee;
use ieee.std_logic_1164.all;

entity full_spec is
    port (
     rb_clock                    : in std_ulogic;
     ack0_new                    : in std_ulogic;
     ack1_new                    : in std_ulogic;
     req0                        : in std_ulogic;
     req1                        : in std_ulogic;
     reset                       : in std_ulogic;
     fails                       : out
                                 std_ulogic_vector(0 to 5)
    );

end full_spec;

architecture rb of full_spec is
    signal not_rb_reset: std_ulogic ;
    signal rb_reset    : std_ulogic ;
```

```
    type rb_enum_type is (iu_enum_slash_err,
          iu_enum_slash_z);

begin

    p:
    process
    begin
     wait until rb_clock'event and rb_clock='1';
     not_rb_reset <= '1';
    end process;
    rb_reset <= reset or (not not_rb_reset);


 f1:
     fails(0) <= not b21((((((not ack0_new) or (not
               ack1_new))) /= '0')
     or (rb_reset /= '0')  or (rb_clock /= '1'));
    f2:
    block
        constant n: integer := 4;
        signal v: std_ulogic_vector(0 to n-1) :=
               "1110";
        signal v_out: std_ulogic_vector(0 to n-1);
        signal vnext: std_ulogic_vector(0 to n-1);
        signal ok: std_ulogic := '1';
    begin
     v_out(0) <= '0';
     v_out(1) <= v(1) and ('1');
     v_out(2) <= v(2) and (((not req0) and (not
               req1)));
     v_out(3) <= v(3) and ((not ((not ack0_new) and
               (not ack1_new))));
```

```
        vnext(0) <= '0';
        vnext(1) <= v_out(1);
        vnext(2) <= v_out(1);
        vnext(3) <= v_out(2);

           p:
           process
        begin
        wait until rb_clock'event and rb_clock='1';
               if rb_reset = '1' then
                   v <= "1110";
                   ok <= '1';
               else
                   if (not (v(3) and (not ((not ack0_new)
                      and (not ack1_new))
      ))) = '0' then
                       ok <= '0';
                   elsif ok = '0' then
                       ok <= '1';
                   end if;

                   v <= vnext;
               end if;
      end process;


          fails(1) <= not b21 ((ok /= '0') or (rb_reset
                      /= '0')
       or (rb_clock /= '1'));
      end block;

      f3:
```

```
block
    constant n: integer := 4;
    signal v: std_ulogic_vector(0 to n-1) :=
           "1110";
    signal v_out: std_ulogic_vector(0 to n-1);
    signal vnext: std_ulogic_vector(0 to n-1);
    signal ok: std_ulogic := '1';
begin
 v_out(0) <= '0';
 v_out(1) <= v(1) and ('1');
 v_out(2) <= v(2) and ((req0 and (not ack0_new)));
 v_out(3) <= v(3) and ((not ack0_new));

 vnext(0) <= '0';
 vnext(1) <= v_out(1);
 vnext(2) <= v_out(1);
 vnext(3) <= v_out(2);

   p:
   process
 begin
 wait until rb_clock'event and rb_clock='1';
        if rb_reset = '1' then
            v <= "1110";
            ok <= '1';
        else
            if (not (v(3) and (not ack0_new))) =
                    '0' then
                ok <= '0';
            elsif ok = '0' then
                ok <= '1';
            end if;
```

```
                v <= vnext;
            end if;
        end process;

5
         fails(2) <= not b21 ((ok /= '0') or (rb_reset
                    /= '0')
         or (rb_clock /= '1'));
        end block;
10
        f4:
        block
            constant n: integer := 4;
            signal v: std_ulogic_vector(0 to n-1) :=
15                  "1110";
            signal v_out: std_ulogic_vector(0 to n-1);
            signal vnext: std_ulogic_vector(0 to n-1);
            signal ok: std_ulogic := '1';
        begin
20       v_out(0) <= '0';
         v_out(1) <= v(1) and ('1');
         v_out(2) <= v(2) and (((not req0) and req1));
         v_out(3) <= v(3) and ((not ack1_new));

25       vnext(0) <= '0';
         vnext(1) <= v_out(1);
         vnext(2) <= v_out(1);
         vnext(3) <= v_out(2);

30          p:
            process
        begin
        wait until rb_clock'event and rb_clock='1';
```

```
            if rb_reset = '1' then
                v <= "1110";
                ok <= '1';
            else
                if (not (v(3) and (not ack1_new))) =
                        '0' then
                    ok <= '0';
                elsif ok = '0' then
                    ok <= '1';
                end if;

                v <= vnext;
            end if;
        end process;


         fails(3) <= not b2l ((ok /= '0') or (rb_reset
                    /= '0')
         or (rb_clock /= '1'));
        end block;

        f5:
        block
            constant n: integer := 4;
            signal v: std_ulogic_vector(0 to n-1) :=
                    "1110";
            signal v_out: std_ulogic_vector(0 to n-1);
            signal vnext: std_ulogic_vector(0 to n-1);
            signal ok: std_ulogic := '1';
        begin
         v_out(0) <= '0';
         v_out(1) <= v(1) and ('1');
         v_out(2) <= v(2) and ((req1 and ack0_new));
```

```
        v_out(3) <= v(3) and ((not ack1_new));

        vnext(0) <= '0';
        vnext(1) <= v_out(1);
        vnext(2) <= v_out(1);
        vnext(3) <= v_out(2);

           p:
           process
        begin
        wait until rb_clock'event and rb_clock='1';
               if rb_reset = '1' then
                   v <= "1110";
                   ok <= '1';
               else
                   if (not (v(3) and (not ack1_new))) =
                            '0' then
                       ok <= '0';
                   elsif ok = '0' then
                       ok <= '1';
                   end if;

                   v <= vnext;
              end if;
      end process;


           fails(4) <= not b2l ((ok /= '0') or (rb_reset
                      /= '0')
       or (rb_clock /= '1'));
      end block;

      f6:
```

```
block
    constant n: integer := 4;
    signal v: std_ulogic_vector(0 to n-1) :=
                "1110";
    signal v_out: std_ulogic_vector(0 to n-1);
    signal vnext: std_ulogic_vector(0 to n-1);
    signal ok: std_ulogic := '1';
begin
 v_out(0) <= '0';
 v_out(1) <= v(1) and ('1');
 v_out(2) <= v(2) and ((req0 and (not req1)));
 v_out(3) <= v(3) and ((not ack0_new));

 vnext(0) <= '0';
 vnext(1) <= v_out(1);
 vnext(2) <= v_out(1);
 vnext(3) <= v_out(2);

   p:
   process
 begin
 wait until rb_clock'event and rb_clock='1';
        if rb_reset = '1' then
            v <= "1110";
            ok <= '1';
        else
            if (not (v(3) and (not ack0_new))) =
                    '0' then
                ok <= '0';
            elsif ok = '0' then
                ok <= '1';
            end if;
```

```
                    v <= vnext;
                end if;
        end process;

5
            fails(5) <= not b21 ((ok /= '0') or (rb_reset
                        /= '0')
         or (rb_clock /= '1'));
        end block;
10
    end rb;
```

APPENDIX C

DEFINITION OF TERMS AND COMPARISON CRITERIA

"Have I written enough properties?" - A method of comparison between specification and implementation

Abstract. This work presents a novel approach for evaluating the quality of the model checking process. Given a model of a design (or implementation) and a temporal logic formula that describes a specification, model checking determines whether the model satisfies the specification. Assume that all specification formulas were successfully checked for the implementation. Are we sure that the implementation is correct? If the specification is incomplete, we may fail to find an error in the implementation. On the other hand, if the specification is complete, then the model checking process can be stopped without adding more specification formulas. Thus, knowing whether the specification is complete may both avoid missed implementation errors and save precious verification time.

The completeness of a specification with respect to a given implementation is determined as follows. The specification formula is first transformed into a tableau. The simulation preorder is then used to compare the implementation model and the tableau model. We suggest four comparison criteria, each revealing a certain dissimilarity between the implementation and the specification. If all comparison criteria are empty, we conclude that the tableau is bisimilar to the implementation model and that the specification fully describes the implementation. We also conclude that there are no redundant states in the implementation.

The method is exemplified on a small hardware example. We implemented our method symbolically as an extension to SMV. The implementation involves efficient OBDD manipulations that reduce the number of OBDD variables from $4n$ to $2n$.

1 Introduction

This work presents a novel approach for evaluating the quality of the model checking process. Given a model of the design (or implementation) and a temporal logic formula that describes a specification, model checking [8, 2] determines whether the model satisfies the specification.

Assume that all specification formulas were successfully checked for the implementation. Are we sure that the implementation is correct? If the specification is incomplete, we may fail to find an error in the implementation. On the other hand, if the specification is complete, then the model checking process can be stopped without checking additional specification formulas. Thus, knowing whether the specification is complete may both avoid missed implementation errors and save precious verification time.

Below we describe our method to determine whether a specification is complete with respect to a given implementation. We restrict our attention to safety properties written in the universal branching-time logic ACTL [3]. This logic is relatively restricted, but can still express most of the specifications used in practice. Moreover, it can fully characterizes every deterministic implementation. We consider a single specification formula (the conjunction of all properties).

We first apply model checking to verify that the specification formula is true for the implementation model. The formula is then transformed into a *tableau* [3]. By definition, since the formula is true for the model, the tableau is greater by the *simulation preorder* [9] than the model.

We defined a *reduced tableau* for ACTL safety formulas. Our tableau is based on the *Particle tableau* for LTL, presented in [6]. We further reduce their tableau by removing redundant tableau states.

We next use the simulation preorder to find differences between the implementation and its specification. For example, if we find a reachable tableau state with no corresponding implementation state, then we argue that one of the two holds. Either the specification is not restrictive enough or the implementation fails to implement a meaningful state. Our method will not be able to determine which of the arguments is correct. However, the evidence for the dissimilarity (in this case a tableau state that none of the implementation states are mapped to) will assist the designer to make the decision.

We suggest four *comparison criteria*, each revealing a certain dissimilarity between the implementation and specification. If all comparison criteria are empty, we conclude that the tableau is bisimilar to the implementation model and that the specification fully describes the implementation. We also conclude that there are no redundant states in the implementation.

The practical aspects of this method are straightforward. Model checking activity in industry executes the following methodology: A verification engineer reads the specification, sets up a work environment and then proceeds to present the model checker with a sequence of properties in order to verify the design correctness [1]. The design (or implementation) on which this activity is executed can be quite large nowadays. As a result the set of properties written and verified becomes large as well, to the point that the engineer loses control over it.

A large property set makes it necessary to construct tools to evaluate its overall quality. The basic question to answer is: "Have I written enough properties?". The current solution is to manually review the property set. However, this solution is not scalable and furthermore since it is done manually it makes the description of model checking as "formal verification" imprecise. This inadequate solution indicates a growing need for tools that may be able to tell the engineer when the design is "bug-free" and therefore cut down development time.

Quality evaluation of verification activity is not new. Traditional verification methods have developed measurement criteria to measure the quality of test suites [11]. This area of verification is called *coverage*. The notion of coverage in model checking is to have a specification that covers the entire functionality

required from the implementation. This can be divided into two questions:

1. Whether the environment is rich enough to provide all possible input sequences.
2. Whether the specification contains a sufficient set of properties.

The method we present addresses both problems as will be later shown by an example.

We compared a small hardware example with the reduced tableau. For the complete specification formula we received a reduced tableau with 20 states. The tableau presented in [3] would have a state space of 2^{15} states for this formula. It is interesting to note that in this example not all the implementation variables are observable.

We implemented our method symbolically as an extension to the symbolic model checker SMV [7]. Given a model with n state variables, a straightforward implementation of this method can create intermediate results that consists of $4n$ OBDD variables. However, our implementation reduces the required number of OBDD variables from $4n$ to $2n$.
The main contributions of our paper can be summarized as follows:

- We suggest for the first time a theoretical framework that provides quality evaluation for model checking. The suggested comparison criteria can assist the designer in finding errors in the design by indicating points in which the design and the specification disagree, and suggest criteria for terminating the verification effort.
- We implemented our method symbolically within SMV. Thus, it can be invoked automatically once the model checking terminates successfully.
  Of a special interest is the symbolic computation of the simulation relation for which no good symbolic algorithm is known.
- We defined a new reduced tableau for ACTL that is often significantly smaller in the number of states and transitions than known tableaux for ACTL.

In these days, another work on coverage of model checking has been independently developed [5]. The work computes the percentage of states in which a change in an observable proposition will not affect the correctness of the specification. Their evidence is closely related to our criterion of *Unimplemented State*. In their paper they list a number of limitations of their work. They are unable to give path evidence, cannot point out functionality missing in the model, and they have no indication that the specification is complete. In the conclusion we explain how our work solves these problems.

The analysis we perform compares the two models and tries to identify dissimilarities. It is therefore related to tautology checking of finite state machines as is done in [10]. However the method in [10] is suggested as an alternative to model checking and not as a complementary method.

The rest of this paper is organized as follows. Section 2 gives the necessary background. Section 3 describes the comparison criteria and the method for their

use. Section 4 exemplifies the different criteria by applying the method to a small hardware circuit. Section 5 presents symbolic algorithms that implement our method. In Section 6 we discuss the reduced tableau for ACTL safety formulas. Finally, the last section describes future work and concludes the paper.

2 Preliminaries

Our specification language is the universal branching-time temporal logic ACTL [3], restricted to safety properties. Let AP be a set of atomic propositions. The set of ACTL *safety formulas* is defined inductively in negation normal form, where negations are applied only to atomic propositions. It consists of the temporal operators $\mathbf{X}$ ("next-state") and $\mathbf{W}$ ("weak until") and the path quantifier $\mathbf{A}$ ("for all paths").

- If $p \in AP$ then both p and $\neg p$ are ACTL safety formulas.
- If φ_1 and φ_2 are ACTL safety formulas then so are $\varphi_1 \wedge \varphi_2$, $\varphi_1 \vee \varphi_2$, $\mathbf{AX}\varphi_1$, and $\mathbf{A}[\varphi_1 \mathbf{W} \varphi_2]$[1].

We use Kripke structures to model our implementations. A *Kripke structure* is a tuple $M = (S, S_0, R, L)$ where S is a finite set of states; $S_0 \subseteq S$ is the set of initial states; $R \subseteq S \times S$ is the transition relation that must be total; and $L : S \to 2^{AP}$ is the labeling function that maps each state to the set of atomic propositions true at that state.

A *path* in M from a state s is a sequence $s_0, s_1, \ldots$ such that $s_0 = s$ and for every i, $(s_i, s_{i+1}) \in R$.

The logic ACTL is interpreted over a state s in a Kripke structure M. The formal definition is omitted here. Intuitively, $\mathbf{AX}\varphi_1$ is true in s if all its successors satisfy φ_1. $\mathbf{A}[\varphi_1 \mathbf{W} \varphi_2]$ is true in s if along every path from s, either φ_1 holds forever or φ_2 eventually holds and φ_1 holds up to that point. We say that a structure M satisfies a formula φ, denoted $M \models \varphi$, if every initial state of M satisfies φ.

Let $M = (S, S_0, R, L)$ and $M' = (S', S'_0, R', L')$ be two Kripke structures over the same set of atomic propositions AP. A relation $SIM \subseteq S \times S'$ is a *simulation preorder* from M to M' [9] if for every initial state s_0 of M there is an initial state s'_0 of M' such that $(s_0, s'_0) \in SIM$. Moreover, if $(s, s') \in SIM$ then the following holds:

- $L(s) = L'(s')$, and
- $\forall s_1[(s, s_1) \in R \Longrightarrow \exists s'_1[(s', s'_1) \in R' \wedge (s_1, s'_1) \in SIM]]$.

If there is a simulation preorder from M to M', we write $M \leq M'$ and say that M *simulates* M'.

It is well known [3] that if $M \leq M'$ then for every ACTL formula φ, if $M' \models \varphi$ then $M \models \varphi$. Furthermore, for every ACTL safety formula ψ it is

[1] Full ACTL includes also formulas of the form $\mathbf{A}[\varphi_1 \mathbf{U} \varphi_2]$ ("strong until").

possible to construct a Kripke structure $T(\psi)$, called a *tableau*[2] for ψ, that has the following *tableau properties* [3].

- $T(\psi) \models \psi$.
- For every structure M, $M \models \psi \iff M \leq T(\psi)$.

Intuitively, the simulation preorder relates two states if the computation tree starting from the state of the smaller model can be embedded in the computation tree starting from the state of the greater one. This, however, is not sufficient in order to determine how similar the two structures are. Instead, we use the *reachable simulation preorder* that relates two states if they are in the simulation preorder and are also reachable from initial states along corresponding paths.

Formally, let $SIM \subseteq S \times S'$ be the *greatest* simulation preorder from M to M'. The *reachable simulation preorder* for SIM, $ReachSIM \subseteq SIM$, is defined by: $(s, s') \in ReachSIM$ if and only if there is a path $\pi = s_0, s_1, \ldots, s_k$ in M with $s_0 \in S_0$ and $s_k = s$ and a path $\pi' = s'_0, s'_1, \ldots, s'_k$ in M' with $s'_0 \in S'_0$ and $s'_k = s'$ such that for all $0 \leq j \leq k$, $(s_j, s'_j) \in SIM$.
In this case, the paths π and π' are called *corresponding paths* leading to s and s'.

Lemma 1. *$ReachSIM$ is a simulation preorder from M to M'.*

The proof of the lemma is postponed to Appendix A.

3 Comparison Criteria

Let $M = (S_i, S_{0i}, R_i, L_i)$ be an implementation structure and $T(\psi) = (S_t, S_{0t}, R_t, L_t)$ be a tableau structure over a common set of atomic propositions AP. For the two structures we consider only reachable states that are the start of an infinite path.

Assume $M \leq T(\psi)$. We define four criteria, each is associated with a set. A criterion is said to hold if the appropriate set is empty. For convenience we name each criterion the same as the appropriate set. The following sets define the criteria :

1. $UnImplementedStartState = \{s_t \in S_{0t} \mid \forall s_i \in S_{0i} [(s_i, s_t) \notin ReachSIM]\}$
   An Unimplemented Start State is an initial tableau state that has no corresponding initial state in the implementation structure. The existence of such a state may indicate that the specification does not properly constrain the set of start states. It may also indicate the lack of a required initial state in the implementation.
2. $UnImplementedState = \{s_t \in S_t \mid \forall s_i \in S_i [(s_i, s_t) \notin ReachSIM]\}$
   An Unimplemented State is a state of the tableau that has no corresponding state in the implementation structure. This difference may suggest that the specification is not tight enough, or that a meaningful state was not implemented.

[2] The tableau for full ACTL is a *fair* Kripke structure (not defined here). It has the same properties except that $\models$ and $\leq$ are defined for fair structures.

3. $UnImplementedTransition = \{(s_t, s'_t) \in R_t \mid \exists s_i, s'_i \in S_i,$
$[(s_i, s_t) \in ReachSIM, (s'_i, s'_t) \in ReachSIM$ and $(s_i, s'_i) \notin R_i]\}$
An Unimplemented Transition is a transition between two states of the tableau, for which a corresponding transition in the implementation does not exist. The existence of such a transition may suggest that the specification is not tight enough, or that a required transition (between reachable implementation states) was not implemented.
4. $ManyToOne = \{s_t \in S_t \mid \exists s_{1i}, s_{2i} \in S_i [(s_{1i}, s_t) \in ReachSIM, (s_{2i}, s_t) \in ReachSIM$ and $s_{1i} \neq s_{2i}]\}$
A Many To One state is a tableau state to which multiple implementation states are mapped. The existence of such a state may indicate that the specification is not detailed enough. It may also suggest that the implementation contains redundancy.

Our criteria are defined for any tableau that has the tableau properties as defined in Section 2. Any dissimilarity between the implementation and the specification will result in a non empty criterion. Empty criteria indicate completeness, but they are hard to obtain on traditional tableaux since such tableaux contain redundancies. In the reduced tableau presented in Section 6, redundancies are removed and therefore empty criteria are more likely to be achieved.
Given a structure M and a property ψ our method consists of the following steps:

1. Apply model checking to verify that $M \models \psi$.
2. Build a (reduced) tableau $T(\psi)$ for ψ.
3. Compute SIM of $(M, T(\psi))$.
4. Compute $ReachSIM$ of $(M, T(\psi))$ from SIM of $(M, T(\psi))$.
5. For each of the comparison criteria, evaluate if its corresponding set is empty and if not present evidence for its failure.

Theorem 2. *Let M be an implementation model and ψ be an ACTL safety formula such that $M \models \psi$. Let $T(\psi)$ be a tableau for ψ that has the tableau properties. If the comparison criteria 1-3 hold then $T(\psi) \leq M$.*

The proof of this theorem is left to Appendix B. The proof implies that if criteria 1-3 hold then $T(\psi)$ and M are in fact *bisimilar*. The fourth criterion is not necessary for completeness since whenever there are several non-bisimilar implementation states that are mapped to the same tableau state, then there is also an unimplemented state or transition. However, this criterion may reveal redundancies in the implementation.

It is important to note that the goal is not to find a smaller set of criteria that guarantees the specification completeness. The purpose of the criteria is to assist the designer in the debugging process. Thus, we are looking for meaningful criteria that can distinguish among different types of problems and identify them. In Section 6 we define an additional criterion that can reveal redundancy in the specification.

4 Example

Consider a synchronous arbiter with two inputs, $req0, req1$ and two outputs $ack0, ack1$. The assertion of ack_i is a response to the assertion of req_i. Initially, both outputs of the arbiter are inactive. At any time, at most one acknowledge output may be active. The arbiter grants one of the active requests in the next cycle, and uses a round robin algorithm in case both request inputs are active. Furthermore in the case of *simultaneous assertion* (i.e. both requests are asserted and were not asserted in the previous cycle), request 0 has priority in the first *simultaneous assertion* occurrence. In any additional occurrence of *simultaneous assertion* the priority rotates with respect to the previous occurrence.
The implementation and the specification will share a common set of atomic propositions $AP = \{req0, req1, ack0, ack1\}$. An implementation of the arbiter M, written in the SMV language is presented below:

```
1)  var
2)    req0, req1, ack0, ack1, robin : boolean;
3)  assign
4)    init(ack0) := 0;
5)    init(ack1) := 0;
6)    init(robin) := 0;
7)  next(ack0) := case
8)    !req0                 : 0;        - No request results no ack
9)    !req1                 : 1;        - A single request
10)   !ack0 & !ack1         : !robin;   - Simultaneous requests assertions
11)   1                     : !ack0;    - Both requesting , toggle ack
12) esac;
13) next(ack1) := case
14)   !req1                 : 0;        - No request results no ack
15)   !req0                 : 1;        - A single request
16)   !ack0 & !ack1         : robin;    - simultaneous assertion
17)   1                     : !ack1;    - Both requesting , toggle ack
18) esac;
19) next(robin) := if req0 & req1 &!ack0 & !ack1 then !robin
20)                        else robin endif;    - Two simultaneous request assertions
```

From the verbal description given at the beginning of the section, one may derive a temporal formula that specifies the arbiter :

$$
\begin{aligned}
\psi = {} & \neg ack0 \wedge \neg ack1 \wedge \\
& \mathbf{A}[(\neg req0 \vee \neg req1 \vee ack0 \vee ack1)\mathbf{W} \\
& \quad (req0 \wedge req1 \wedge \neg ack0 \wedge \neg ack1 \wedge \mathbf{AX} ack0)] \qquad \wedge \qquad - \varphi_0 \\
& \mathbf{AG}(\\
& \quad (\neg ack0 \vee \neg ack1) \qquad \wedge \qquad - \varphi_1
\end{aligned}
$$

$$
\begin{array}{lll}
(\neg req0 \wedge \neg req1 \rightarrow \mathbf{AX}(\neg ack0 \wedge \neg ack1)) & \wedge & -\varphi_2 \\
(req0 \wedge \neg req1 \rightarrow \mathbf{AX}ack0) & \wedge & -\varphi_3 \\
(\neg req0 \wedge req1 \rightarrow \mathbf{AX}ack1) & \wedge & -\varphi_4 \\
(req1 \wedge ack0 \rightarrow \mathbf{AX}ack1) & \wedge & -\varphi_5 \\
(req0 \wedge ack1 \rightarrow \mathbf{AX}ack0) & \wedge & -\varphi_6 \\
(req0 \wedge req1 \wedge \neg ack0 \wedge \neg ack1 \rightarrow \mathbf{AX}(ack0 \rightarrow & & \\
\mathbf{A}[(\neg req0 \vee \neg req1 \vee ack0 \vee ack1)\mathbf{W} & & \\
\quad (req0 \wedge req1 \wedge \neg ack0 \wedge \neg ack1 \wedge \mathbf{AX}ack1)])) & \wedge & -\varphi_7 \\
(req0 \wedge req1 \wedge \neg ack0 \wedge \neg ack1 \rightarrow \mathbf{AX}(ack1 \rightarrow & & \\
\mathbf{A}[(\neg req0 \vee \neg req1 \vee ack0 \vee ack1)\mathbf{W} & & \\
\quad (req0 \wedge req1 \wedge \neg ack0 \wedge \neg ack1 \wedge \mathbf{AX}ack0)])) &) & -\varphi_8
\end{array}
$$

where $\mathbf{AG}\varphi \equiv \mathbf{A}[\varphi \mathbf{W} false]$. We verified that $M \models \psi$ using the SMV model checker. We then applied our method. We found that all comparison criteria hold. We therefore concluded that ψ is a complete specification for M.
In order to exemplify the ability of our method we changed the implementation and the specification in different ways. In all cases the modified implementation satisfied the modified specification. However, our method reported the failure of some of the criteria. By examining the evidence supplied by the report, we could detect flaws in either the implementation or the specification.

4.1 Unimplemented Transition evidence

Consider a modified version of the implementation M, named M_{trans} obtained by adding the line : $robin \,\&\, ack1 : \{0,1\}$;
between line (10) and line (11), and by adding the line :
$ack1 : !next(ack0)$; between line (16) and line (17).
Consider also the modified formula ψ_{trans} obtained from ψ by replacing φ_6 with:
$(req0 \wedge ack1 \rightarrow \mathbf{AX}(ack0 \vee ack1)) \wedge$
SMV shows that $M_{trans} \models \psi_{trans}$. However, applying the comparison method on M_{trans} and ψ_{trans}, reports an *Unimplemented Transition*. It supplies as an evidence a transition between tableau states s_t and s'_t such that $L_t(s_t) = L_t(s'_t) = \{req0, req1, !ack0, ack1\}$. Such a transition is possible by ψ_{trans} but not possible in M_{trans} in case variable *robin* is not asserted.

If we check the reason for the incomplete specification we note that the evidence shows a cycle with $req0$ and $ack1$ asserted followed by a cycle were $ack1$ is asserted. This ill behavior violates the round robin requirement. The complete specification would detect that M_{trans} has a bug, since $M_{trans} \not\models \psi$.

4.2 Unimplemented state evidence

Consider a modified version of the implementation M, named M_{unimp} obtained by adding line : $ack0 : \{0,1\}$;
between lines (10) and line (11), and replacing line (2) with the following lines:
2.1) $req0\_temp, req1, ack0, ack1, robin : boolean$;

2.2) $define\ req0 := req0\_temp \& !(ack0 \& ack1)$;
Here $req0\_temp$ is a free variable, and the input $req0$ is a restricted input such that if the state satisfies $ack0\&ack1$ then $req0$ is forced to be inactive.
Consider also the modified formula ψ_{unimp} obtained from ψ by deleting φ_1. SMV shows that $M_{unimp} \models \psi_{unimp}$. However, applying the comparison method on M_{unimp} and ψ_{unimp}, reports an *Unimplemented State*. It supplies as an evidence the state s_t such that $L_t(s_t) = \{req0, !req1, ack0, ack1\}$. This state is possible by ψ_{unimp} but not possible in M_{unimp}.
If we check the source of the incomplete specification we note that the evidence violates the mutual exclusion property. Both of the arbiter outputs $ack0$ and $ack1$ are active. The complete specification would detect that M_{unimp} has a bug, since $M_{unimp} \not\models \psi$.
Note that in this example we can also identify that $req0$ in M_{unimp} is a restricted input relative to the formula ψ_{unimp}. The state space of M_{unimp} does not include the states $\{req0, req1, ack0, ack1\}$ or $\{req0, !req1, ack0, ack1\}$. A restricted environment may hide bugs, so this is just as important as finding missing properties.

4.3 Many To One evidence

A nonempty *Many To One* criterion may imply one of two cases. Redundant implementation, or incompleteness. The latter case is always accompanied with one of criteria 1-3. The former case where criteria 1-3 hold but we have a *Many To One* evidence implies that the implementation is complete with respect to the specification, but it is not efficient and contains redundancies. There is a smaller implementation that can preserve the completeness. This information may give insight on the efficiency of the implementation.
The following implementation M_{m2o} uses 5 implementation variables and two free inputs instead of 3 variables and two inputs of implementation M. Criteria 1-3 are met for M_{m2o} with respect to ψ.

```
1) var
2)   req0, req1, req0q, req1q, ack0q, ack1q, robin : boolean;
3) assign
4)   init(req0q) := 0; init(req1q) := 0;
5)   init(ack0q) := 0; init(ack1q) := 0;
6)   init(robin) := 1;
7) define
8)   ack0 := case
9)     !req0q                  : 0;      - No request results no ack
10)    !req1q                  : 1;      - A single request
11)    !ack0q & !ack1q         : !robin; - Simultaneous requests assertions
12)    1                       : !ack0q; - Both requesting , toggle ack
13)  esac;
14)  ack1 := case
```

```
15)   !req1q                    : 0;        - No request results no ack
16)   !req0q                    : 1;        - A single request
17)   !ack0q & !ack1q           : robin;    - simultaneous assertion
18)   1                         : !ack1q;   - Both requesting , toggle ack
19) esac;
20) assign
21)  next(robin) := if req0 & req1 &!ack0 & !ack1 then !robin
22)    else robin endif;                    - Two simultaneous request assertions
23)  next(req0q) := req0; next(req1q) := req1;
24)  next(ack0q) := ack0; next(ack1q) := ack1;
```

Applying model checking will show that $M_{m2o} \models \psi$.
In the above example we keep information of the current inputs $req0$ and $req1$, as well as their value in the previous cycle (i.e. $req0q$ and $req1q$). Intuitively, this duplicates each state in M to four states in the state space of M_{m2o}.

4.4 Unimplemented Start State evidence

The *Unimplemented Start State* criterion does not hold when the specification is not restricted to the valid start states. Consider a specification formula obtained from ψ by removing the φ_0 subformula. Applying the comparison method on M and the modified formula would yield a *Unimplemented Start State* evidence of a tableau state s_{0t} such that $\{ack0, !ack1\} \subseteq L_t(s_{0t})$. Restricting the specification to the valid start states would cause the *Unimplemented Start State* criteria to hold.

4.5 Non-Observable Implementation Variables

As can be seen in this example, a state of the implementation is not uniquely determined by $req0$, $req1$, $ack0$ and $ack1$. The variable *robin* effects the other variables, but it is a non-observable intermediate variable. This variable is not explicitly described in the specification, and does not appear in the common set of atomic propositions AP, referred to by the simulation preorder. Our criteria are defined with respect to observable variables only, but are not limited to systems where all the variables are observable.

5 Implementation of the Method

5.1 Symbolic Algorithms

In this section we present the symbolic algorithms that implement various parts of our method. In particular, we show how to compute symbolically the simulation relation. Our implementation will require less memory than the naive implementation since we reduce the number of OBDD variables. In Section 5.2 we show how this is achieved.

For conciseness, we use $R(s, s')$, $S(s)$ etc. instead of $(s, s') \in R$, $s \in S$.

Computing SIM: Let $M = (S_i, S_{0i}, R_i, L_i)$ be the implementation structure and let $T(\psi) = (S_t, S_{0t}, R_t, L_t)$ be a tableau structure. The following pseudo-code depicts the algorithm for computing SIM:
Init: $SIM_0(s_i, s_t) := \{ (s_i, s_t) \in S_i \times S_t \mid L_i(s_i) = L_t(s_t) \}$; $j := 0$
Repeat {
$SIM_{j+1} := \{ (s_i, s_t) \mid \forall s_i' [R_i(s_i, s_i') \rightarrow \exists s_t' [R_t(s_t, s_t') \wedge SIM_j(s_i', s_t')]] \wedge SIM_j(s_i, s_t)\}$
$j := j + 1$ } *until* $SIM_j = SIM_{j-1}$
$SIM := SIM_j$

Computing $ReachSIM$: Given the simulation relation SIM of the pair $(M, T(\psi))$ the following pseudo-code depicts the algorithm for computing $ReachSIM$:
Init: $ReachSIM_0 := (S_{0i} \times S_{0t}) \cap SIM$; $j := 0$
Repeat {
$ReachSIM_{j+1} := ReachSIM_j \cup$
$\{ (s_i', s_t') \mid \exists s_i, s_t (ReachSIM_j(s_i, s_t) \wedge R_i(s_i, s_i') \wedge R_t(s_t, s_t') \wedge SIM(s_i', s_t')) \}$
$j := j + 1$ } *until* $ReachSIM_j = ReachSIM_{j-1}$
$ReachSIM := ReachSIM_j$

5.2 Efficient OBDD Implementation

We now turn our attention to improving the performance of the algorithms described in the previous section. We assume that an implementation of such an algorithm will be done within a symbolic model checker such as SMV [7]. Since formal analysis always suffers from state explosion it is necessary to find methods to efficiently utilize computer memory. When working with OBDDs one possible way to do so is to try to minimize the number of OBDD variables that any OBDD created during the computation will have.

We can see from the algorithms presented before that some of the sets, constructed in intermediate computation steps, are defined over four sets of states: implementation states, specification states, tagged (next) implementation states, and tagged (next) specification states. For example, the computation of SIM_{j+1} is defined by means of the implementation states s_i, specification states s_t, tagged implementation states s_i' (representing implementation next states), and tagged specification states s_t' (representing specification next states).

Assume that we need at most n bits to encode each set of states. Then potentially some of the OBDDs created in the intermediate computations will have $4n$ OBDD variables. However, by breaking the algorithm operations to smaller ones and manipulating OBDDs in a nonstandard way we managed to bound the number of variables of the OBDDs created in intermediate computations by $2n$.

We define two operations, *compose* and *compose_odd*, that operate on two OBDDs a and b over a total number of $3n$ variables. As explained later, the main advantage of these operations is that they can be implemented using only $2n$ OBDD variables.

$$compose(\mathbf{y}, \mathbf{u}) \equiv \exists \mathbf{x}(a(\mathbf{x}, \mathbf{y}) \wedge b(\mathbf{x}, \mathbf{u})) \quad (1)$$

$$compose\_odd(\mathbf{y}, \mathbf{u}) \equiv \exists \mathbf{x}(a(\mathbf{y}, \mathbf{x}) \wedge b(\mathbf{u}, \mathbf{x})). \quad (2)$$

SIM and $ReachSIM$ can be implemented using *compose* and *compose_odd* as follows. Let $\mathbf{v_i}$, $\mathbf{v'_i}$ be the encoding of the states s_i, s'_i respectively. Similarly, let $\mathbf{v_t}$, $\mathbf{v'_t}$ be the encoding of s_t, s'_t respectively.

$SIM_{j+1}(\mathbf{v_i}, \mathbf{v_t}) := SIM_j(\mathbf{v_i}, \mathbf{v_t}) \wedge$
$\neg compose\_odd(R_i(\mathbf{v_i}, \mathbf{v'_i}), \neg compose\_odd(R_t(\mathbf{v_t}, \mathbf{v'_t}), SIM_j(\mathbf{v'_i}, \mathbf{v'_t})))$

$ReachSIM_{j+1}(\mathbf{v'_i}, \mathbf{v'_t}) := ReachSIM_j(\mathbf{v'_i}, \mathbf{v'_t}) \vee$
$(compose(compose(ReachSIM_j(\mathbf{v_i}, \mathbf{v_t}), R_i(\mathbf{v_i}, \mathbf{v'_i})), R_t(\mathbf{v_t}, \mathbf{v'_t})) \wedge SIM(\mathbf{v'_i}, \mathbf{v'_t}))$

The derivation of these expressions can be found in Appendix C. The algorithms above require that the implementation and specification "step" together along the transition relation. We break this to stepping along one, followed by stepping along the other. This is possible since transitions of the two structures are independent.

The comparison criteria *Unimplemented Transition* and *Many To One* can also be implemented with these operations. The two other criteria are defined over $2n$ variables and do not require such manipulation.

$UnimplementedTransition(\mathbf{v_t}, \mathbf{v'_t}) := R_t(\mathbf{v_t}, \mathbf{v'_t}) \wedge$
$compose(compose(\neg R_i(\mathbf{v_i}, \mathbf{v'_i}), ReachSIM(\mathbf{v_i}, \mathbf{v_t})), ReachSIM(\mathbf{v'_i}, \mathbf{v'_t}))$

$ManyToOne(\mathbf{v_t}) :=$
$\exists \mathbf{v_1}(ReachSIM(\mathbf{v_1}, \mathbf{v_t}) \wedge compose((\mathbf{v_1} \neq \mathbf{v_2}), ReachSIM(\mathbf{v_2}, \mathbf{v_t})))$

The details of these derivations can be found in Appendix C.

Up to now we showed how to reduce the number of OBDD variables from $4n$ to $3n$. We now show how to further reduce this number to $2n$. Our first step is to use the same OBDD variables to represent the implementation variables $\mathbf{v_i}$ and the specification variables $\mathbf{v_t}$. These OBDD variables will be referred to as *untagged*. Similarly, we use the same OBDD variables to represent $\mathbf{v'_i}$ and $\mathbf{v'_t}$. They will be referred to as *tagged* OBDD variables.

We also specify that whenever we have relations over both implementation variables and specification variables then the implementation variables are represented by untagged OBDD variables while the specification variables are represented by tagged OBDD variables. Note that now the relations R_i, R_t, SIM, $ReachSIM$ are all defined over the same sets of OBDD variables. Consequently, in all the derived expressions we apply *compose* and *compose_odd* to OBDDs that share variables, i.e. $\mathbf{y}$ and $\mathbf{u}$ are represented by the same OBDD variables. The implementation of *compose* and *compose_odd* uses non-standard OBDD operations in such a way that the resulting OBDDs are also defined over the same $2n$ variables.

Notice that this requires that the OBDD variable change semantics in the result (e.g., in Equation 1 $\mathbf{y}$ is represented by tagged OBDD variables in the input parameters and by untagged variables in the result). OBDD packages can easily be extended with these operations.

6 Reduced Tableau and Redundancies in Specification

6.1 Smaller tableau structure

When striving for completeness, the size of tableau structures as defined in [3] is usually too large to be practical, and may be much larger than the state space of the given implementation. This is because the state space of such tableaux contain all combinations of subformulas of the specification formula. Such tableaux usually contain many redundant states, that can be removed while preserving the tableau properties. If not removed, these states may introduce evidences which are not of interest.

Much of the redundancies can be eliminated if each state contains *exactly* the set of formulas required for satisfying the specification formula. Consider for example the ACTL formula $\mathbf{AXAX}p$. Its set of subformulas is $\{\mathbf{AXAX}p, \mathbf{AX}p, p\}$. We desire a tableau structure in which each state contains only the set of subformulas required to satisfy the formula. In this case, the initial state should satisfy $\mathbf{AXAX}p$, its successor should satisfy $\mathbf{AX}p$ and its successor should satisfy p. In each of these states all unmentioned subformulas have a "don't care" value. Thus, one state of the reduced tableau represents many states. For instance, the initial state $\{\mathbf{AXAX}p\}$ represents four initial states in the traditional tableau [3]. In such examples we may get a linear size tableau instead of an exponential one.

Following the above motivation, the reduced tableau will be defined over a *3-value labeling* for atomic propositions, i.e., for an atomic proposition p, a state may be labeled by either p, $\neg p$ or neither of them. Also, only the reachable portion of the structure will be constructed.

Further reduction may be obtained if the set of successors for each state is constructed more carefully. If a state s has two successors s' and s'', such that the set of formulas of s'' is contained in the set of formulas of s', then s' is not constructed. Any tableau behavior starting at s' has a corresponding behavior from s''. Thus, it is unnecessary to include both.

Given an ACTL safety formulas, the definition of the reduced tableau is derived from the Particle tableau for LTL, presented in [6] by replacing the use of the $\mathbf{X}$ temporal operator by $\mathbf{AX}$. Since the only difference between LTL and ACTL is that temporal operators are always preceded by the universal path quantifier, this change is sufficient. In general, we will obtain a smaller tableau since we also avoid the construction of redundant successors.

Since the reduced tableau is based on the 3-value labeling, the definition of satisfaction and simulation preorder are changed accordingly. Our reduced tableau $T(\psi)$ for ACTL then has the same properties as the one in [3]:

- $T(\psi) \models \psi$.
- For every Kripke structure M, $M \leq T(\psi)$ if and only if $M \models \psi$.

Note that adopting the reduced tableau also requires modifications to our criteria due to the 3-value labeling semantics.

6.2 Reduced Tableau Results

We have defined the reduced tableau and proved its tableau properties. In addition we have adapted the comparison criteria to comply with the *3-value labeling*. We also coded the reduced tableau construction and the comparison criteria into the SMV model checker, performing the structure comparison in a symbolic manner.

We have run the arbiter example of Section 4 with the reduced tableau. For the complete specification formula ψ presented there we received a structure with 20 states. A traditional tableau structure would have a state space of 2^{15} states for ψ.

6.3 Identifying redundancies in the specification

Section 3 defines criteria that characterize when a specification is rich enough (i.e., complete). We would like also to determine whether a complete specification contains redundancies, i.e., subformulas that can be removed or be rewritten without destroying the completeness of the specification.

Given the reduced tableau, we suggest a new criterion, called *One To Many*, that identifies implementation states that are mapped (by $ReachSIM$) to multiple tableau states. Finding such states means that there is a smaller structure that corresponds to an equivalent specification formula. The criterion $OneToMany$ is defined by:
$OneToMany = \{s_i \in S_i \mid \exists s_{1t}, s_{2t} \in S_t[(s_i, s_{1t}) \in ReachSIM \wedge (s_i, s_{2t}) \in ReachSIM \wedge s_{1t} \neq s_{2t}]\}$.

6.4 One To Many Example

The following example demonstrates the One to Many criterion. It identifies a redundant sub formula, which does not add to the completeness of the specification formula. Consider the following specification formula :

$$
\begin{array}{lll}
\psi_{One2Many} = & & \\
\quad \neg ack0 \wedge \neg ack1 & \wedge & \\
\quad \mathbf{A}[(\neg req0 \vee \neg req1 \vee ack0 \vee ack1)\mathbf{W} & & \\
\quad\quad (req0 \wedge req1 \wedge \neg ack0 \wedge \neg ack1 \wedge \mathbf{AX}ack0)] \wedge & & -\ \varphi_0 \\
\quad \mathbf{AG}(& & \\
\quad\quad (\neg ack0 \vee \neg ack1) & \wedge & -\ \varphi_1 \\
\quad\quad (\neg req0 \wedge \neg req1 \wedge \mathbf{AX}(\neg ack0 \wedge \neg ack1) & \vee & -\ \varphi_2 \\
\quad\quad req0 \wedge \neg req1 \wedge \mathbf{AX}ack0 & \vee & -\ \varphi_3 \\
\quad\quad \neg req0 \wedge req1 \wedge ack1 \wedge \mathbf{AX}ack1 & \vee & -\ \varphi_4 \\
\quad\quad req0 \wedge req1 \wedge ack0 \wedge \mathbf{AX}ack1 & \vee & -\ \varphi_5 \\
\quad\quad req0 \wedge req1 \wedge ack1 \wedge \mathbf{AX}ack0 & \vee & -\ \varphi_6 \\
\quad\quad req1 \wedge ack1 \wedge \mathbf{AX}(ack0 \wedge \neg ack1) & \vee & -\ \varphi_{redundant} \\
\quad\quad req0 \wedge req1 \wedge \neg ack0 \wedge \neg ack1 \wedge \mathbf{AX}(& & \\
\quad\quad ack0 \wedge \mathbf{A}[(\neg req0 \vee \neg req1 \vee ack0 \vee ack1)\mathbf{W} & & \\
\quad\quad (req0 \wedge req1 \wedge \neg ack0 \wedge \neg ack1 \wedge \mathbf{AX}ack1)] \vee & & \\
\quad\quad ack1 \wedge \mathbf{A}[(\neg req0 \vee \neg req1 \vee ack0 \vee ack1)\mathbf{W} & & \\
\quad\quad (req0 \wedge req1 \wedge \neg ack0 \wedge \neg ack1 \wedge \mathbf{AX}ack0)])))) & & -\ \varphi_7
\end{array}
$$

Our method reported that for $\psi_{One2Many}$ criteria 1-4 are met. In addition, it reported that the *One To Many* criterion is not met. As an evidence it provides the implementation state s_i such that $L_i(s_i) = \{req0, req1, \neg ack0, ack1\}$. This state is mapped to s_{1t} and s_{2t} of the reduced tableau for which $L_t(s_{1t}) = \{req0, req1, \neg ack0, ack1\}$ and $L_t(s_{2t}) = \{req1, \neg ack0, ack1\}$.
We may note that $\varphi_{redundant}$ sub formulas agrees with φ_6 for states labeled with $\{req0, req1\}$, and does not agree with φ_4 for states labeled with $\{\neg req0, req1\}$. Since it comes as a disjunct, it does not limit the reachable simulation, and does not add allowed behavior. Deleting sub formula $\varphi_{redundant}$ leaves a specification formula such that criteria 1-4 are met and the *One to Many* criterion is also met.

7 Future work

In this paper we presented a novel approach for evaluating the quality of the model checking process. The method we described can give an engineer the confidence that the model is indeed "bug-free" and reduce the development time.

We are aware that the work we have done is not complete. There are a few technical issues that will have to be addressed:

1. **State explosion:** The state explosion problem is even more acute than with model checking because we have to perform symbolic computations while M and $T(\psi)$ are both in memory. This implies that at present the circuits that we can apply this method to are smaller than those that we can model check. Therefore we currently cannot provide a solution for large models. However we believe that over time optimizations in this area will be introduced as

4. T. A. Henzinger, O. Kupferman, and S. K. Rajamani. Fair simulation. In *Proc. of the 7th Conference on Concurrency Theory (CONCUR'97)*, volume 1243 of *LNCS*, Warsaw, July 1997.
5. Hoskote, Kam, Ho, and Zhao. Coverage estimation for symbolic model checking. In *proceedings of the 36rd Design Automation Conference (DAC'99)*. IEEE Computer Society Press, June 1999.
6. Z. Manna and A. Pnueli. *Temporal verifications of Reactive Systems - Safety.* Springer-Verlag, 1995.
7. K. L. McMillan. *The SMV System DRAFT.* Carnegie Mellon University, Pittsburgh, PA, 1992.
8. K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic Press, Norwell, MA, 1993.
9. R. Milner. An algebraic definition of simulation between programs. In *In proceedings of the 2nd International Joint Conference on Artificial Intelligence*, pages 481–489, September 1971.
10. T. Filkorn. A method for symbolic verification of synchronous circuits. In D. Borrione and R. Waxman, editors, *Proceedings of The Tenth International Symposium on Computer Hardware Description Languages and their Applications*, IFIP WG 10.2, pages 249–259, Marseille, April 1991. North-Holland.
11. Elaine J. Weyuker and Bingchiang Jeng. Analyzing partition testing strategies. *IEEE Transactions on Software Engineering*, 2(17), July 1991.

A Proof of Lemma 1

Lemma 1 *$ReachSIM$ is a simulation preorder from M to M'.*

Proof. Clearly, for initial states, $(s_0, s'_0) \in SIM$ if and only if $(s_0, s'_0) \in ReachSIM$. Thus, for every initial state of M there is a $ReachSIM$-related initial state of M'. Let $(s, s') \in ReachSIM$. First we note that since $ReachSIM \subseteq SIM$, $(s, s') \in SIM$ and therefore $L(s) = L'(s')$.

Now let $(s, s_1) \in R$. Then there is s'_1 such that $(s', s'_1) \in R'$ and $(s_1, s'_1) \in SIM$. Since $(s, s') \in ReachSIM$, there are corresponding paths π and π' leading to s and s'. These paths can be extended to corresponding paths leading to s_1 and s'_1. Thus, $(s_1, s'_1) \in ReachSIM$. □

B Proof of Theorem 2

Theorem 1 *Let M be an implementation model and ψ be an ACTL safety formula such that $M \models \psi$. Let $T(\psi)$ be a tableau for ψ that satisfy the tableau properties. If the comparison criteria 1-3 hold then $T(\psi) \leq M$.*

Proof. Since $M \models \psi$, $M \leq T(\psi)$. Thus, there is a simulation preorder $SIM \subseteq S_i \times S_t$. Let $ReachSIM$ be the reachable simulation preorder for SIM. Then $ReachSIM^{-1} \subseteq S_t \times S_i$ is defined by $(s_t, s_i) \in ReachSIM^{-1}$ if and only if $(s_i, s_t) \in ReachSIM$. We show that $ReachSIM^{-1}$ is a simulation preorder from $T(\psi)$ to M.

was done for model checking. We are investigating the possibility of running this method separately on small properties and then combining the results.
Another solution to the state explosion is to compute the criteria "on-the-fly" together with the computation of *ReachSIM* and to discover violations before *ReachSIM* is fully computed.
A third solution is to use the algorithm in [5] as a preliminary step, and try to expand it to fully support our methodology. The definition of Unimplemented State is closely related to the evidences in [5]. On the other hand, our Unimplemented Transition criterion provides path evidences, while path coverage is not addressed by the methodology of [5]. Furthermore, our method can indicate that the specification and the implementation totally agree. This may serve as an indication that the verification process can be stopped.

2. **Irrelevant information:** Similar to the area of traditional simulation coverage, measurement of quality produces a lot of information which is often irrelevant. A major problem is that specifications tend to be incomplete by nature and therefore we do not necessarily want to achieve a bisimulation relation between the specification and implementation. Therefore, it will eventually be necessary to devise techniques to filter the results such that only the interesting evidences are reported.
We are also investigating whether the reduced tableau described in Section 6 is optimal in the sense that it does not contain any redundancies.

3. **Expressivity:** Our specification language is currently restricted to ACTL safety formulas. It is straight forward to extend our method to full ACTL. This will require, however, to add fairness constraints to the tableau structure and to use the *fair simulation preorder* [3]. Unfortunately, there is no efficient algorithm to implement fair simulation [4]. Thus, it is currently impractical to use full ACTL. There is a need to find logics that are both reasonable in terms of their expressivity and practical in terms of tableau construction and comparison criteria.

Acknowledgment: We thank Ilan Beer for suggesting to look into the problem of coverage in model checking. The first author thanks Galileo Technology for the opportunity to work on the subject.

References

1. I. Beer, S. Ben-David, C. Eisner, and A. Landver. Rulebase - an industry oriented formal verification tool. In *33th Design Automation Conference*, 1996. DAC.
2. E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT press, 1999. To appear.
3. O. Grumberg and D.E. Long. Model checking and modular verification. *ACM Trans. on Programming Languages and Systems*, 16(3):843–871, 1994.

Let s_{0t} be an initial state of $T(\psi)$. Since $UnImplementedStartState$ is empty, there must be an initial state s_{0i} of M such that $(s_{0i}, s_{0t}) \in ReachSIM$. Thus, $(s_{0t}, s_{0i}) \in ReachSIM^{-1}$.
Now let $(s_t, s_i) \in ReachSIM^{-1}$. Since $(s_i, s_t) \in ReachSIM$, $L_t(s_t) = L_i(s_i)$. Let $(s_t, s'_t) \in R_t$. Since $UnimplementedState$ is empty, there must be a state $s'_i \in S_i$ such that $(s'_i, s'_t) \in ReachSIM$. Since $UnImplementedTransition$ is empty we get $(s_i, s'_i) \in R_i$. Thus, s'_i is a successor of s_i and $(s'_t, s'_i) \in ReachSIM^{-1}$. We conclude that $ReachSIM^{-1}$ is a simulation preorder and therefore $T(\psi) \leq M$. □

Note that, since $ReachSIM$ and $ReachSIM^{-1}$ are both simulation preorders, $ReachSIM$ is actually a bisimulation relation.

C Derivation of the *compose* formulas

Following is the algebraic derivation that enables the use of the *compose* and *compose_odd* operations described in Section 5.

- Derivation of SIM_j:
  $SIM_{j+1}(\mathbf{v_i}, \mathbf{v_t}) :=$
  $\forall \mathbf{v'_i}[R_i(\mathbf{v_i}, \mathbf{v'_i}) \rightarrow \exists \mathbf{v'_t}\,[\,R_t(\mathbf{v_t}, \mathbf{v'_t}) \wedge SIM_j(\mathbf{v'_i}, \mathbf{v'_t})]] \wedge SIM_j(\mathbf{v_i}, \mathbf{v_t}) =$
  $\forall \mathbf{v'_i}[\neg R_i(\mathbf{v_i}, \mathbf{v'_i}) \vee \exists \mathbf{v'_t}\,[\,R_t(\mathbf{v_t}, \mathbf{v'_t}) \wedge SIM_j(\mathbf{v'_i}, \mathbf{v'_t})]] \wedge SIM_j(\mathbf{v_i}, \mathbf{v_t}) =$
  $\neg\exists \mathbf{v'_i}[R_i(\mathbf{v_i}, \mathbf{v'_i}) \wedge \neg\exists \mathbf{v'_t}\,[\,R_t(\mathbf{v_t}, \mathbf{v'_t}) \wedge SIM_j(\mathbf{v'_i}, \mathbf{v'_t})]] \wedge SIM_j(\mathbf{v_i}, \mathbf{v_t}) =$
  $\neg compose\_odd(R_i(\mathbf{v_i}, \mathbf{v'_i}), \neg compose\_odd(R_t(\mathbf{v_t}, \mathbf{v'_t}), SIM_j(\mathbf{v'_i}, \mathbf{v'_t}))) \wedge SIM_j(\mathbf{v_i}, \mathbf{v_t})$

- Derivation of $ReachSIM_j$:
  $f_{j+1}(\mathbf{v_t}, \mathbf{v'_i}) :=$
  $\exists \mathbf{v_i}(ReachSIM_j(\mathbf{v_i}, \mathbf{v_t}) \wedge R_i(\mathbf{v_i}, \mathbf{v'_i})) = compose(ReachSIM_j(\mathbf{v_i}, \mathbf{v_t}), R_i(\mathbf{v_i}, \mathbf{v'_i}))$
  $g_{j+1}(\mathbf{v'_i}, \mathbf{v'_t}) :=$
  $\exists \mathbf{v_t}(f_{j+1}(\mathbf{v_t}, \mathbf{v'_i}) \wedge R_t(\mathbf{v_t}, \mathbf{v'_t})) = compose(f_{j+1}(\mathbf{v_t}, \mathbf{v'_i}), R_t(\mathbf{v_t}, \mathbf{v'_t}))$
  $g_{j+1}(\mathbf{v_i}, \mathbf{v_t}) := g_{j+1}(\mathbf{v'_i}, \mathbf{v'_t})$
  $ReachSIM_{j+1}(\mathbf{v_i}, \mathbf{v_t}) := (g_{j+1}(\mathbf{v_i}, \mathbf{v_t}) \wedge SIM(\mathbf{v_i}, \mathbf{v_t})) \vee ReachSIM_j(\mathbf{v_i}, \mathbf{v_t})$

- Derivation of $ManyToOne$:
  $ManyToOne(\mathbf{v_t}) :=$
  $\exists \mathbf{v_1}, \mathbf{v_2}(ReachSIM(\mathbf{v_1}, \mathbf{v_t}) \wedge ReachSIM(\mathbf{v_2}, \mathbf{v_t}) \wedge (\mathbf{v_1} \neq \mathbf{v_2})) =$
  $\exists \mathbf{v_1}(ReachSIM(\mathbf{v_1}, \mathbf{v_t}) \wedge \exists \mathbf{v_2}((\mathbf{v_2} \neq \mathbf{v_1}) \wedge ReachSIM(\mathbf{v_2}, \mathbf{v_t}))) =$
  $\exists \mathbf{v_1}(ReachSIM(\mathbf{v_1}, \mathbf{v_t}) \wedge compose((\mathbf{v_1} \neq \mathbf{v_2}), ReachSIM(\mathbf{v_2}, \mathbf{v_t})))$

- Derivation of $UnimplementedTransition$:
  $f(\mathbf{v'_i}, \mathbf{v_t}) :=$
  $\exists \mathbf{v_i}(\neg R_i(\mathbf{v_i}, \mathbf{v'_i}) \wedge ReachSIM(\mathbf{v_i}, \mathbf{v_t})) = compose(\neg R_i(\mathbf{v_i}, \mathbf{v'_i}), ReachSIM(\mathbf{v_i}, \mathbf{v_t}))$
  $g(\mathbf{v_t}, \mathbf{v'_t}) :=$
  $\exists \mathbf{v'_i}(f(\mathbf{v'_i}, \mathbf{v_t}) \wedge ReachSIM(\mathbf{v'_i}, \mathbf{v'_t})) = compose(f(\mathbf{v'_i}, \mathbf{v_t}), ReachSIM(\mathbf{v'_i}, \mathbf{v'_t}))$
  $UnimplementedTransition(\mathbf{v_t}, \mathbf{v'_t}) := g(\mathbf{v_t}, \mathbf{v'_t}) \wedge R_t(\mathbf{v_t}, \mathbf{v'_t})$

CLAIMS

1. A method for verification, comprising:

providing an implementation model, which defines model states of a target system and model transitions between the model states;

providing a specification of the target system, comprising properties that the system is expected to obey;

creating a tableau from the specification, the tableau defining tableau states with tableau transitions between the tableau states in accordance with the properties; and

comparing the tableau transitions to the model transitions to determine whether a discrepancy exists therebetween.

2. A method according to claim 1, wherein creating the tableau comprises defining a finite state machine using a hardware description language.

3. A method according to claim 2, wherein the implementation model has model inputs and outputs, and wherein defining the finite state machine comprises describing a virtual device having inputs and outputs corresponding to the model inputs and outputs of the implementation model.

4. A method according to claim 3, wherein comparing the transitions comprises performing a reachability analysis using both the implementation model and the tableau while providing identical inputs to the inputs of both the implementation model and the tableau, and verifying that the outputs are always identical.

5. A method according to claim 4, wherein performing the reachability analysis comprises comparing the model and the tableau automatically using a model checker.

6. A method according to claim 4, wherein performing the reachability analysis comprises providing evidence of a tableau transition that is not implemented in the model.

7. A method according to claim 1, wherein comparing the tableau transitions comprises associating model transitions with corresponding tableau transitions.

8. A method according to claim 7, wherein associating the transitions comprises defining a reachable simulation preorder relating the model and the tableau.

9. A method according to claim 7, wherein associating the transitions comprises finding a tableau transition that is not implemented in the model.

10. A method according to claim 9, wherein finding the tableau transition that is not implemented in the model comprises deriving an indication, based on the unimplemented transition, that the specification is not complete with respect to the model.

11. A method according to claim 9, wherein finding the tableau transition that is not implemented in the model comprises deriving an indication, based on the unimplemented transition, that a transition of the target system is missing in the model.

12. A method according to claim 1, and comprising associating model states with corresponding tableau states.

13. A method according to claim 12, wherein associating the model states with the corresponding tableau states comprises finding a tableau state that is not implemented in the model.

14. A method according to claim 13, wherein finding the tableau state that is not implemented in the model comprises deriving an indication, based on the unimplemented state, that the specification is not complete with respect to the model.

15. A method according to claim 13, wherein finding the tableau state that is not implemented in the model comprises deriving an indication, based on the unimplemented state, that a state of the target system is missing in the model.

16. A method according to claim 12, wherein associating the model states with the corresponding tableau states comprises finding multiple model states corresponding to a single tableau state.

17. A method according to claim 1, wherein creating the tableau comprises creating a reduced tableau from which one or more redundant states have been eliminated.

18. A method according to claim 1, wherein comparing the transitions comprises verifying that the specification is a complete and correct description of the implementation model responsive to the comparison.

19. A verification processor, which is configured to receive an implementation model, defining model states of a target system and model transitions between the model states, and to receive a specification of the target system, including properties that the system is

expected to obey, and which is operative to create a tableau from the specification, the tableau defining tableau states with tableau transitions between the tableau states in accordance with the properties, and to compare the tableau transitions to the model transitions to determine whether a discrepancy exists therebetween.

20. A processor according to claim 19, which is operative to perform model checking of the implementation model.

21. A computer software product for verification of a specification of a target system, which specification includes properties that the system is expected to obey, by comparison with an implementation model, which defines model states of the target system and model transitions between the model states, the product comprising a computer-readable medium having computer program instructions recorded therein, which instructions, when read by a computer, cause the computer to create a tableau from the specification, the tableau defining tableau states with tableau transitions between the tableau states in accordance with the properties, and to compare the tableau transitions to the model transitions to determine whether a discrepancy exists therebetween.

22. A product according to claim 21, wherein the program instructions cause the computer to compare the tableau with the model by running a reachability analysis using both the implementation model and the tableau while providing identical inputs to the inputs of both the implementation model and the tableau, and verifying that the outputs are always identical.

23. A product according to claim 22, wherein the reachability analysis is performed using an automatic model checker.

24. A product according to claim 21, wherein the instructions cause the computer to verify that the specification is a complete description of the implementation model.

25. A method for verification, comprising:

providing an implementation model, which defines model states of a target system and model transitions between the model states;

providing a specification of the target system, comprising properties that the system is expected to obey;

creating a tableau from the specification, the tableau defining tableau states with tableau transitions between the tableau states in accordance with the properties; and

comparing the model and the tableau by inputting the model and the tableau to an automatic model checking program.

26. A method according to claim 25, wherein creating the tableau comprises defining a finite state machine using a hardware description language.

27. A method according to claim 26, wherein the input model has model inputs and outputs, and wherein defining the finite state machine comprises describing a virtual device having inputs and outputs corresponding to the model inputs and outputs of the implementation model.

28. A method according to claim 27, wherein comparing the model and the tableau comprises running the model checker while providing identical inputs to the inputs of both the implementation model and the tableau, and verifying that the outputs are always identical.

29. A method according to claim 25, wherein comparing the model and the tableau comprises providing evidence of a transition or state in the tableau that is not implemented in the model.

30. A method according to claim 29, wherein providing the evidence comprises providing a counter-example indicative of the unimplemented transition or state.

31. Model checking apparatus, which is configured to receive an implementation model, defining model states of a target system and model transitions between the model states, and to receive a specification of the target system, including properties that the system is expected to obey, and which is operative to create a tableau from the specification, the tableau defining tableau states with tableau transitions between the tableau states in accordance with the properties, and to compare the tableau to the model by inputting the model and the tableau to an automatic model checking program.

32. A computer software product for verification of a specification of a target system, which specification includes properties that the system is expected to obey, by comparison with an implementation model, which defines model states of the target system and model transitions between the model states, the product comprising a computer-readable medium having computer

program instructions recorded therein, which instructions, when read by a computer, cause the computer to create a tableau from the specification, the tableau defining tableau states with tableau transitions between the tableau states in accordance with the properties, and to compare the tableau to the model by inputting the model and the tableau to an automatic model checking program.

For the applicants,

Sanford T. Colb & Co.

C: 35277

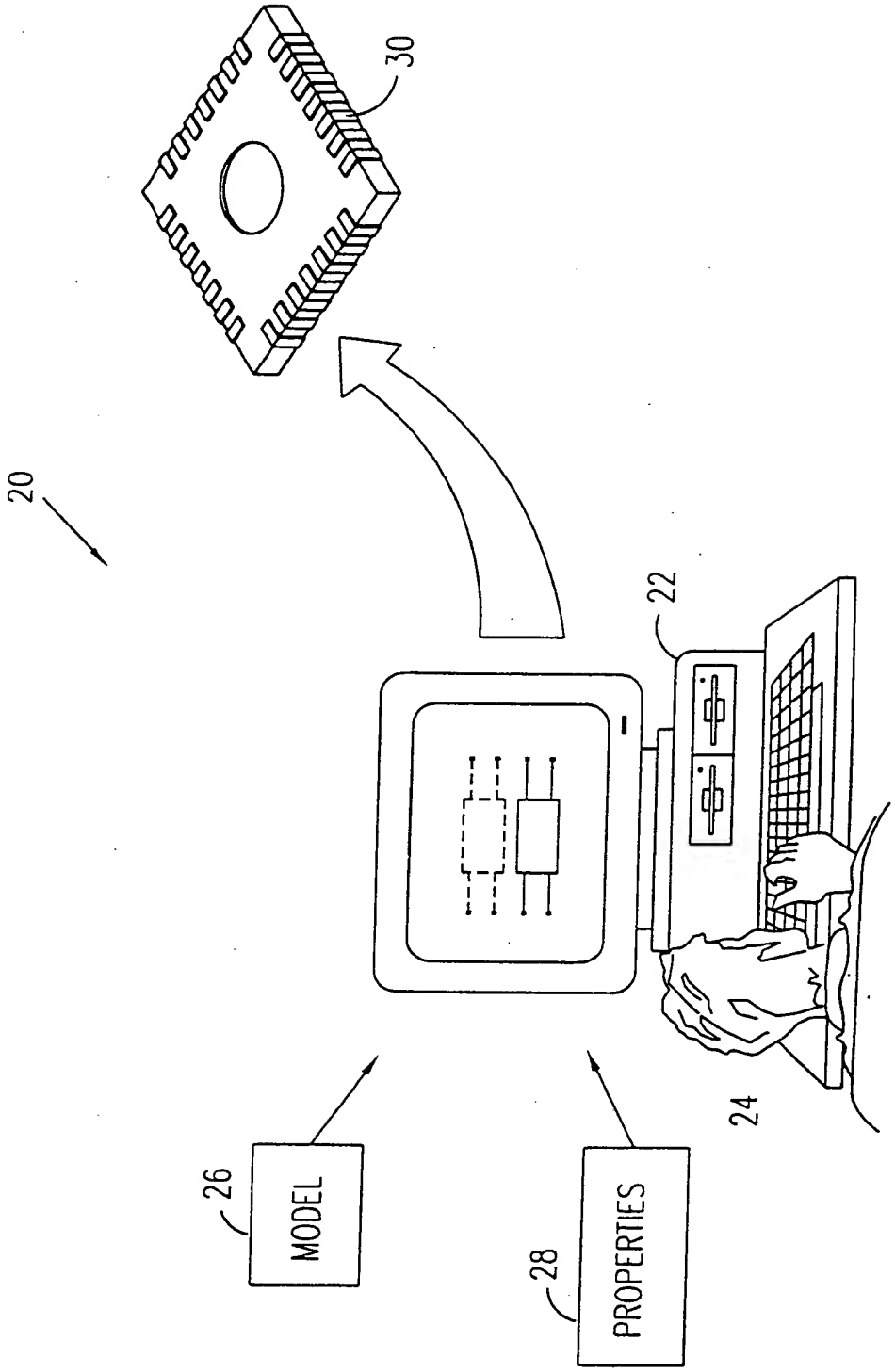


FIG. 1

FIG. 2

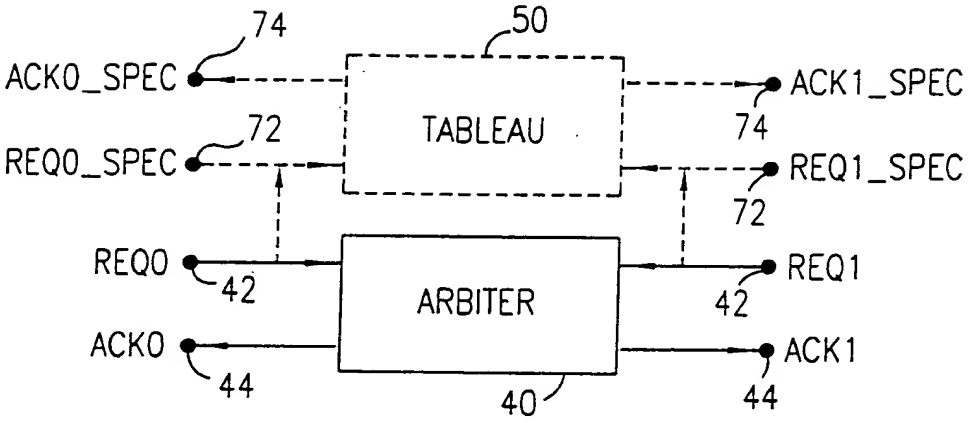

74
50
ACK0_SPEC
TABLEAU
ACK1_SPEC
74
72
REQ0_SPEC
REQ1_SPEC
72
REQ0
42
ARBITER
42
REQ1
ACK0
44
ACK1
40
44

FIG. 3

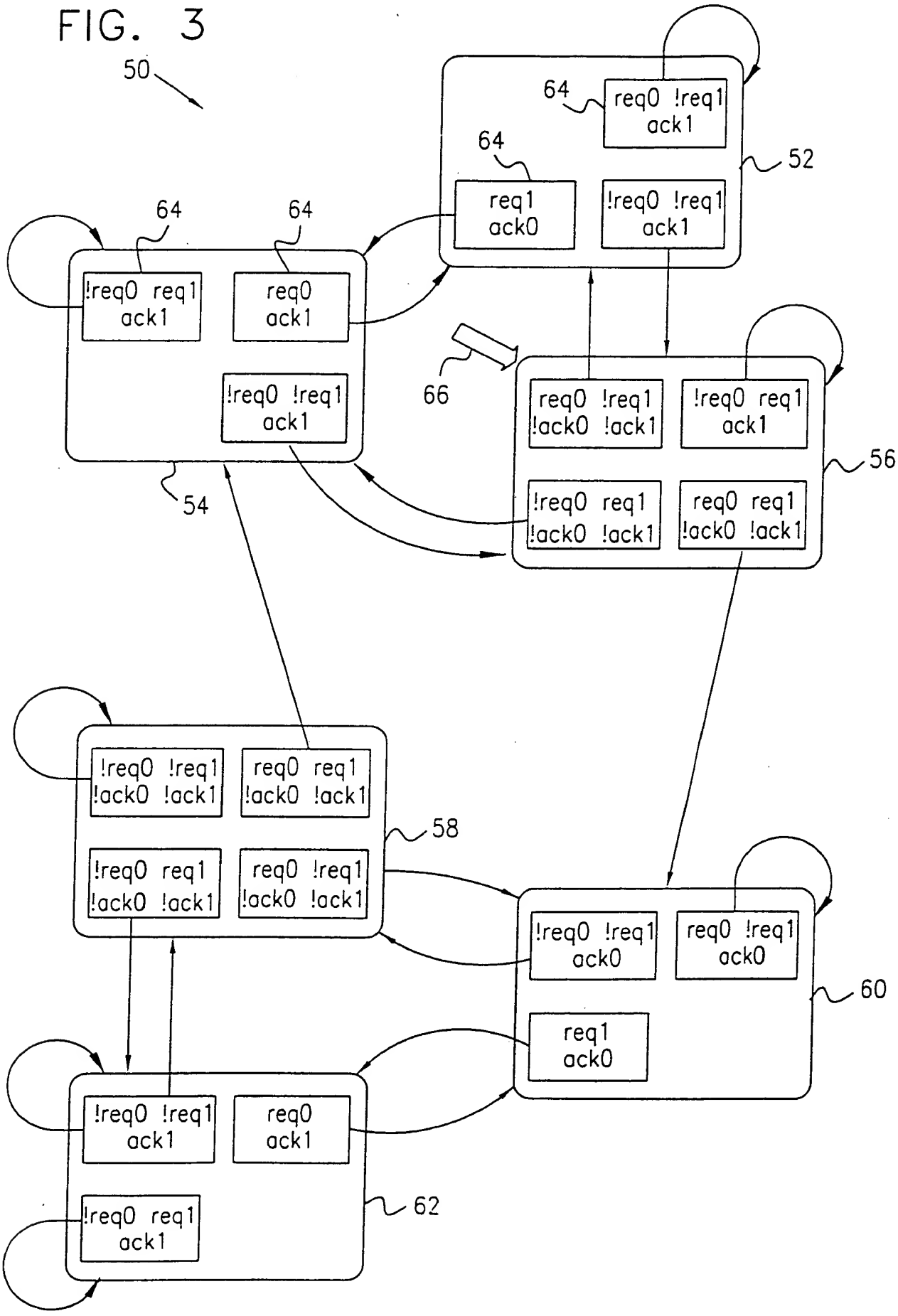
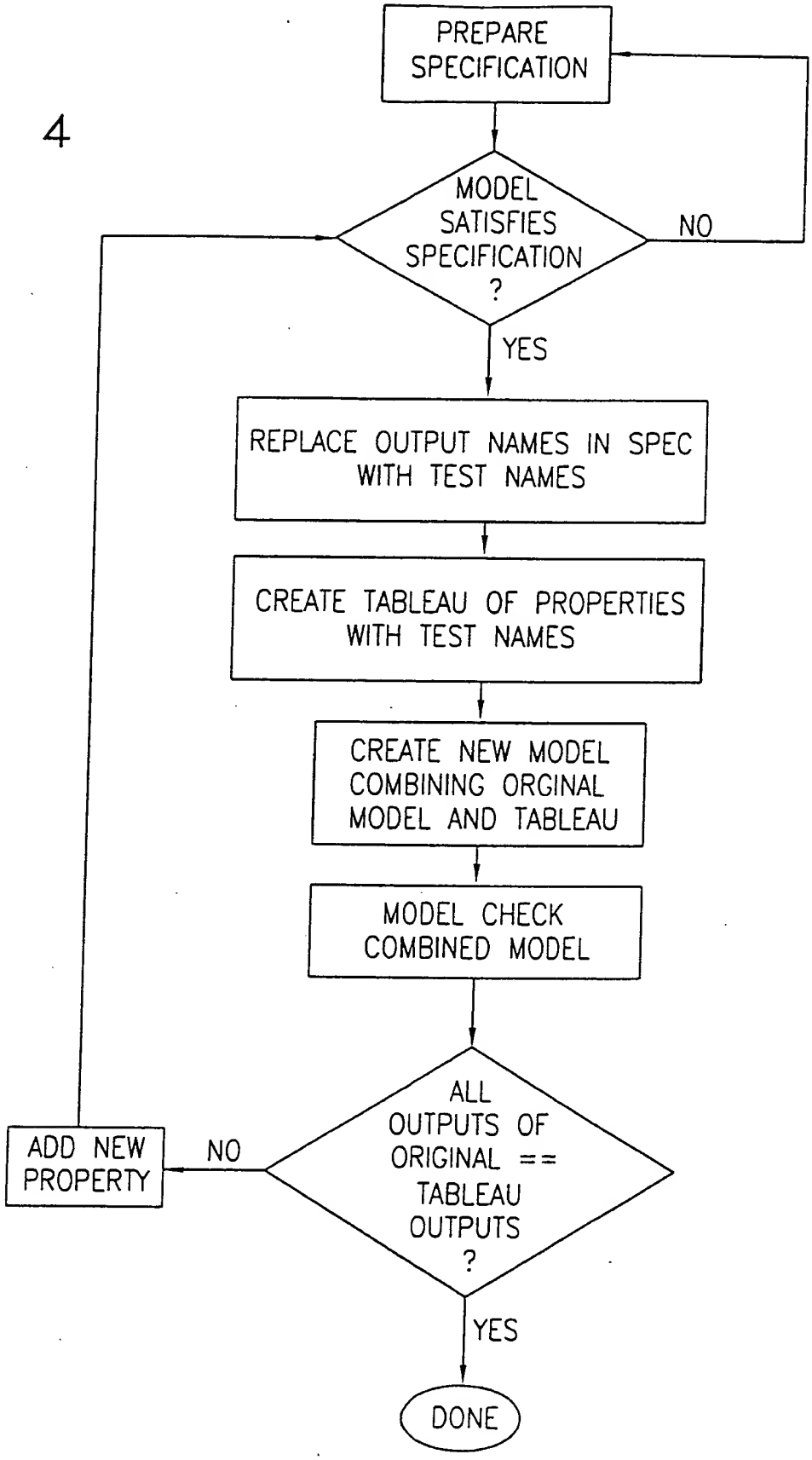

50
64
req0 !req1 ack1
64
req1 ack0
!req0 !req1 ack1
52
64
64
!req0 req1 ack1
req0 ack1
!req0 !req1 ack1
54
66
req0 !req1 !ack0 !ack1
!req0 req1 ack1
!req0 req1 !ack0 !ack1
req0 req1 !ack0 !ack1
56
!req0 !req1 !ack0 !ack1
req0 req1 !ack0 !ack1
!req0 req1 !ack0 !ack1
req0 !req1 !ack0 !ack1
58
!req0 !req1 ack0
req0 !req1 ack0
req1 ack0
60
!req0 !req1 ack1
req0 ack1
!req0 req1 ack1
62

FIG. 4


PREPARE SPECIFICATION
MODEL SATISFIES SPECIFICATION ?
NO
YES
REPLACE OUTPUT NAMES IN SPEC WITH TEST NAMES
CREATE TABLEAU OF PROPERTIES WITH TEST NAMES
CREATE NEW MODEL COMBINING ORGINAL MODEL AND TABLEAU
MODEL CHECK COMBINED MODEL
ALL OUTPUTS OF ORIGINAL == TABLEAU OUTPUTS ?
NO
ADD NEW PROPERTY
YES
DONE

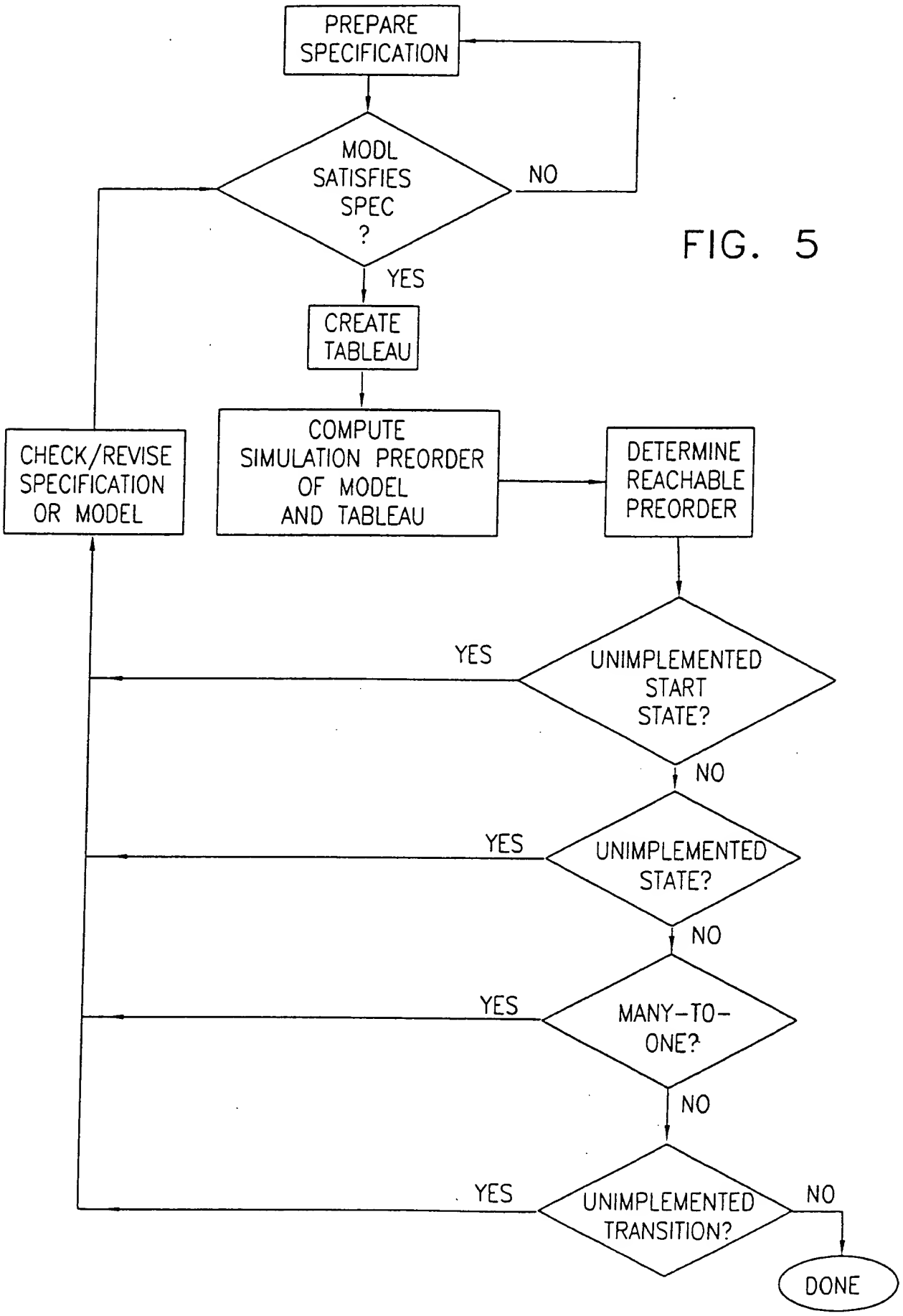

PREPARE SPECIFICATION
MODL SATISFIES SPEC ?
NO
YES
CREATE TABLEAU
COMPUTE SIMULATION PREORDER OF MODEL AND TABLEAU
DETERMINE REACHABLE PREORDER
CHECK/REVISE SPECIFICATION OR MODEL
UNIMPLEMENTED START STATE?
YES
NO
UNIMPLEMENTED STATE?
YES
NO
MANY-TO-ONE?
YES
NO
UNIMPLEMENTED TRANSITION?
YES
NO
DONE

FIG. 5